DULLIN-RETZLAFF
SCHNEIDER-STRASSENBURG

MICRO APPLICATION 17

AMSTRAD

TRUCS ET ASTUCES II POUR LES CPC 464, 664 ET 6128

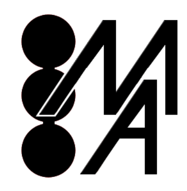


UN LIVRE DATA BECKER



AMSTRAD

TRUCS ET ASTUCES II POUR LES CPC 464, 664 ET 6128



UN LIVRE DATA BECKER

Distribué par : MICRO APPLICATION
13, Rue Sainte Cécile
75009 PARIS

et

EDITION RADIO
3, Rue de l'Eperon
75006 PARIS

ISBN : 2-86899-038-X

Traduction Française assurée par Olivier PAULETTE

Collection dirigée par Mr Philippe OLIVIER
Edition réalisée par Frédérique BEAUDONNET

TABLE DES MATIERES

PREMIERE PARTIE

DEUXIEME PARTIE

TROISIEME PARTIE

INTRODUCTION

COMPOSITION DE L'OUVRAGE

Ce livre est composé de trois parties qui traitent de thèmes différents :

La première partie est consacrée à l'interpréteur BASIC du CPC. Vous y trouverez des programmes et des routines utiles et intéressantes, ainsi que quelques trucs pratiques en BASIC, qui concernent par exemple la protection ou l'accélération des programmes.

A la fin de cette partie, vous trouverez une description précise des ordres BASIC spécifiques au CPC avec des informations sur leurs rôles. Dans la seconde partie de l'ouvrage se trouvent des routines en langage machine qui peuvent être utilisées telles quelles et qui constituent des extensions très pratiques de l'interpréteur BASIC.

Bien que les programmes présentés dans cette partie soient écrits en langage machine, ils présentent un intérêt incontestable, mê me pour les lecteurs qui ne comprennent pas l'assembleur, car ils fonctionnent tels quels. Cette partie propose des programmes prêts à l'emploi, mais vous y trouverez aussi des informations de fond très utiles.

La troisième partie de ce livre est consacrée au langage machine du CPC. Elle contient une liste importante de routines intéressantes du système d'exploitation et de l'interpréteur BASIC. De plus, vous y découvrirez quelques trucs et astuces pour optimiser les programmes en assembleur et pour utiliser certains mécanismes spécifiques du système d'exploitation du CPC, comme par exemple l'extension des instructions avec RSX. Le chapitre 23 décrit une méthode à notre connaissance non encore développée pour relocaliser les programmes en langage machine (déplacement avec adaptation des adresses absolues).

Ce chapitre a été particulièrement détaillé afin d'expliquer les techniques utilisées pour transformer un logiciel et pour manipuler astucieusement la pile par programme. Même les programmeurs avancés pourront sans doute tirer de ce chapitre quelques conseils pour programmer en assembleur avec style.

Bien que la troisième partie soit presque entièrement consacrée au langage machine, il était nécessaire qu'elle intéresse également le lecteur non averti. Ils apprécieront en particulier les deux premiers chapitres pour leur aspect progressif et parce qu'ils comportent des exemples de programmes clairs et généraux.

LA NOTATION HEXADECIMALE

Les nombres hexadécimaux sont précisés par le signe '&'. C'est la notation utilisée par le BASIC du CPC et nous l'avons choisie pour des raisons de standardisation. Certains assembleurs utilisent des notations différentes (par exemple précédé par un '#' ou succédé par un 'H'). Dans ce cas naturellement, il faudra vous conformer à votre assembleur lors de la frappe des programmes.

CARACTERES SPECIAUX DANS LES LISTINGS

Dans cet ouvrage, vous trouverez souvent le caractère '^' Ne le cherchez pas sur votre clavier! Il représente la flèche vers le haut qui sert pour élever à la puissance en BASIC.

LE CLAVIER DES CPC

Malheureusement, les caractères sur certaines touches des claviers des trois CPC ne sont pas les mêmes. Lorsque nous parlons de la touche RETURN dans ce livre (présente sur le 6128), il s'agit de la grosse touche ENTER sur les 664 et 464.

PREMIERE PARTIE

TRUCS ET ASTUCES EN BASIC

I. TRI DE DONNEES

Pourquoi ce thème ?

Ne s'agit-il pas de programmes qui sont difficiles à lire et dont l'utilité est discutable ?

Il est particulièrement intéressant de trouver des algorythmes de tri dont la vitesse est performante. Une grande partie du temps machine utilisé par les ordinateurs est consacrée au tri. Il n'existe pratiquement aucun programme pour utilisateur qui n'utilise pas au moins un peu une fonction de tri. Ce n'est pas sans raison si l'on recherche des routines de tri toujours meilleures, c'est à dire plus rapides.

A l'origine, ce chapitre devait être très court, mais l'étude de ce thème s'est avérée tellement passionnante que nous l'avons ralongé.
Les algorythmes de tri en particulier sont pleins de trucs et leur programmation ne prend que quelques lignes. Pour des programmes simples de moins de 10 lignes, il est étonnant de constater combien les différences peuvent être sensibles. La rapidité ne réside pas dans le fait d'écrire un programme dans n'importe quel langage mais dans le développement d'une routine aussi performante que possible pour le tri de données.

Deux facteurs sont déterminants pour la rapidité du tri :

1. Le nombre de comparaisons à effectuer.

2. Le nombre moyen de déplacement de données, par exemple pour inverser leur place.

La qualité d'un algorythme tient à ces facteurs. Le problème est que le temps de tri n'est pas proportionnel au nombre de données. Cela signifie que :

Lorsque le tri de dix données dure une seconde, la durée du tri de vingt données n'est pas obligatoirement le double. Avec de mauvais algorythmes, la durée aurait plut8t tendance à augmenter au carré. Pour deux fois plus de données, on a besoin de quatre fois plus de temps (4=2*2), pour dix fois plus de données, on a besoin de cent fois plus de temps (100=10*10).

Expliquons maintenant le tri le plus simple, le 'tri à bulles'. Le tri à bulle tire son nom du fait que les données les plus grandes remontent au cours du tri comme les bulles dans l'eau. A la fin on obtient un ensemble trié. Pour celà, on trie les éléments deux à deux. Si l'élément dont l'indice est le plus petit est plus grand, les données sont échangées et le plus grand élément remonte comme une bulle. Puis on recommence l'opération avec l'élément suivant, etc. Cette opération s'arrête seulement lorsque l'ensemble à été complètement remanié. Alors, le plus grand élément se retrouve tout en haut à sa place.

Si notre ensemble comporte n éléments, alors il faudra faire n-1 comparaisons de la sorte.

Puisque le plus grand élément est à sa place, il n'est plus besoin de s'en occuper. Nous ferons donc comme si l'ensemble ne comportait plus que n-1 éléments et nous recommençons l'opération. Après la seconde passe, le deuxième élément par la taille est bien placé. Nous avons donc utilisé n-2 opérations, c'est à dire en tout (n-1)+(n-2). La taille de l'ensemble à trier est alors réduite de 1 et on recommence l'opération jusqu'à ce que tout soit trié. En tout (n-1)+(n-2)+(n-3)+...+2+1=n*(k-1)/2 comparaisons ont été réalisées. Pour n=10 il faudra 45 comparaisons, pour n=100, donc pour 10 fois plus, il faudra 4950 comparaisons, donc plus de 100 fois plus. Le tri augmente au carré !

Pour tous les programmes qui suivent, les conventions suivantes sont retenues :

1. L'ensemble à trier se trouve en a(anz).

2. L'indice maximum de l'ensemble est stocké dans 'anz'. Ainsi, le nombre d'éléments à trier est anz-1, et pour l'indice 0, un élément est mémorisé.

3. Après le tri, l'ensemble trié se retrouve de nouveau en a(anz).

4. Toutes les variables qui ne sont pas précisées par '$' ou '!' sont entière, pour des raisons de gain de temps.

Voici maintenant le programme de base qui concerne la préparation du tri.

```
10 ' tri
20 DEFINT a-z:DEFREAL t
30 RANDOMIZE TIME
40 aus=0:'label pour affichage du tri
50 auga=0:'numero du canal pour l'affichage
60 anz=20:anz=anz-1:'taille de l'ensemble
70 DIM a(anz+1),b(anz),l(20),r(20)
80 FOR i=0 TO anz:a(i)=INT(100*RND):b(i)=a(i):NEXT:'creation
 de l'ensemble a trier
90 READ j$:IF j$="#" THEN END ELSE j=VAL(j$):PRINT j,:IF aus
 THEN PRINT:'lecture des numeros de programme
100 FOR i=0 TO anz:a(i)=b(i):NEXT:t=TIME
110 ON j GOSUB 200,300,400,500,600,700,800,900,1050,1200
120 PRINT#auga,(TIME-t)/300
130 GOTO 90:' caractere de controle de fin
140 DATA 1,2,3,4,5,6,7,8,9,10:'programmes en memoire
150 DATA #
160 REM routine d'affichage du tri
170 FOR n=0 TO anz:PRINT #auga,USING "##";a(n);:PRINT #auga,
"  ";:NEXT
180 PRINT#auga
190 RETURN
```

Et voici le tri à bulles:

```
200 REM tri a bulles
210 FOR i=anz TO 1 STEP -1
220 FOR j=1 TO i
230 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s
240 IF aus THEN GOSUB 170
250 NEXT j,i
260 RETURN
```

La boucle externe réduit l'ensemble à trier à chaque itération. Dans la boucle interne, tous les éléments de l'ensemble restant sont comparés et échangés.

Pour améliorer encore le temps, il faudra supprimer la ligne 240. Elle sert à afficher l'ensemble après chaque comparaison. L'affichage est exécuté si la variable a la valeur -1. Ainsi vous pouvez suivre le déroulement du tri.

Voici quelques développements possibles:

Créez un compteur pour le nombre de comparaisons et un autre pour le nombre d'échanges. Comparez la valeur donnée avec le temps écoulé et avec la valeur théorique. Voyez combien la part du temps d'échanges est importante dans le temps total de travail. (voir aussi le chapitre : accélération des programmes BASIC)

Le tri à bulle est un mauvais type de tri !

Mais avec cet exemple, vous pouvez voir combien les petites transformations peuvent améliorer un algorythme.
Une des faiblesses du tri à bulles tient à ce qu'il trie même ce qui a déjà été trié. Cela peut être aisément constaté. Si un échange est effectué, posons un pointeur. Si à la fin de l'itération le pointeur n'a pas été mis en place, alors le tri aura été fait.

Testez combien cette amélioration aura été efficace du point de vue du temps et du nombre de comparaisons et d'échanges.

L'idée du pointeur peut être encore développée. Au lieu de mettre en place le pointeur, mémorisons l'indice de l'échange. Après une itération, le pointeur contient l'indice du dernier échange. Tous les éléments à l'indice plus élevé sont alors correctement triés et nous n'avons besoin de trier l'ensemble que jusqu'à cet indice.

```
300 REM tri a bulles etendu
310 FOR i=anz TO 1 STEP -1
320 maxj=0
330 FOR j=1 TO i
340 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s:maxj=j
350 IF aus THEN GOSUB 170
360 NEXT j
370 IF maxj=0 THEN RETURN ELSE i=maxj
380 NEXT i
390 RETURN
```

Etudiez ici encore les améliorations avec la méthode expliquée précédemment.

Essayez d'écrire une version qui trie vers le haut, c'est à dire où le plus grand élément remonte et qui trie vers le bas, c'est à dire où le plus petit élément redescent. Utilisez deux pointeurs qui indiquent les limites de l'ensemble non encore trié. Testez les capacités de ce système.

Nous en avons fini avec le tri à bulles. Voici maintenant quelques algorythmes de tri simples, qui sont assez différents. Faites des tests de comparaison entre les différents algorythmes.

Pour mieux comprendre le fonctionnement, vous trouverez l'option d'affichage dans chaque programme. Pour faire les tests de vitesses, il vaut mieux supprimer cette ligne.

TRI DU MAXIMUM

Le tri du maximum recherche le plus grand élément dans l'ensemble non encore trié. Si un élément est plus grand que le plus grand déjà trouvé, les deux éléments sont échangés. Après chaque itération, l'ensemble restant est réduit de un.

```
400 REM tri du maximum
410 FOR i=anz TO 1 STEP -1
420 FOR j=0 TO i-1
430 IF a(j)>a(i) THEN s=a(j):a(j)=a(i):a(i)=s
440 IF aus THEN GOSUB 170
450 NEXT j,i
460 RETURN
```

TRI DIRECT OU TRI SELECTIF

Cette routine est un tri du maximum amélioré. L'amélioration est que l'échange n'est pas automatique quand on trouve un plus grand élément. Au contraire, On recherche d'abord le maximum dans l'ensemble restant puis on l'échange. Le nombre de comparaisons est le même mais il y a moins d'échanges.

```
500 REM tri direct
510 FOR i=anz TO 1 STEP -1:m=0
520 FOR j=0 TO i
530 IF a(j)>m THEN m=a(j):k=j
540 NEXT
550 a=a(i):a(i)=a(k):a(k)=a
560 IF aus THEN GOSUB 170
570 NEXT
580 RETURN
```

TRI PAR INSERTION

Le tri par insertion est issu d'une méthode que chacun de nous connaît. Dans un jeu de carte, nous trions une nouvelle carte en l'insérant parmi celles déjà triées. Le tri par insertion fonctionne exactement de la même manière: La nouvelle carte est déclarée carte la plus grande. Puis elle est comparée avec la carte voisine et si elle est plus petite, elles sont échangées. On commence avec deux éléments puis l'ensemble trié devient de plus en plus grand.

```
600 REM tri par insertion
610 FOR i=1 TO anz
620 FOR j=i TO 1 STEP -1
630 IF a(j)>=a(j-1) THEN 670
640 s=a(j-1):a(j-1)=a(j):a(j)=s
650 IF aus THEN GOSUB 170
660 NEXT j
670 NEXT i
680 RETURN
```

Etant donné qu'un déplacement est plus rapide qu'un échange, le tri par insertion peut lui aussi être amélioré: On recherche d'abord l'emplacement où le nouvel élément trié devra être placé puis on déplace tous les éléments plus grands et on insère enfin le nouvel élément.

```
700 REM tri par insertion ameliore
710 FOR i=1 TO anz
720 a=a(i)
730 FOR j=i-1 TO 0 STEP -1
740 IF a<a(j) THEN NEXT j
750 FOR k=i TO j+2 STEP -1:a(k)=a(k-1):NEXT k
760 a(j+1)=a
770 IF aus THEN GOSUB 170
780 NEXT i
790 RETURN
```

Mais même cette version peut être transformée. Le déplacement de l'ensemble peut être encore plus rapide pour les petits ensemble en rajoutant une boucle.

```
800 REM 3. Version du tri par insertion
810 FOR i=1 TO anz
820 a=a(i):j=i-1
830 IF a>=a(j) THEN 860
840 a(j+1)=a(j):j=j-1
850 IF j>=0 THEN 830
860 a(j+1)=a
870 IF aus THEN GOSUB 170
880 NEXT
890 RETURN
```

TRI BINAIRE

Dans le dernier programme, une grande partie du temps était passée à rechercher la place d'un nouvel élément dans une liste déjà triée. Le temps de cette opération peut être considérablement réduit en utilisant la recherche binaire. Dans le tri par insertion simple, chaque élément était indistinctement comparé. Au contraire, le tri binaire utilise une recherche 'intelligente'. Imaginez que vous devez deviner un nombre entre 1 et 128 en posant des questions. Avec la méthode du tri par insertion, les questions seraient:

le nombre est-il 128?
Non!
le nombre est-il 127?
Non!
le nombre est-il 126?
.
.
.

Il serait plus rentable de demander :

Le nombre est-il supérieur, inférieur ou égal à 64?
supérieur à 64!

La question suivante serait alors :

Le nombre est-il supérieur, inférieur ou égal à 96 ?
inférieur à 96 !

Puis :

Le nombre est-il supérieur, inférieur ou égal à 84 ?
inférieur à 84 !

Le principe de ce questionnaire est d'estimer le nombre recherché. Avec les questions systématiques, on fixe des limites à la réponse. Une méthode de questionnaire semblable est utilisée par une émission de télévision bien connue.

Question: 'Le candidat est-il un homme ou une femme ?'

Réponses possibles, normalement : homme ou femme.

Dans ce cas, il n'existe que deux réponses possibles. Mais il important de noter que le domaine des questions possibles se réduit considérablement en fonction des réponses.

L'intervalle dans lequel peut se situer le nombre est divisé par deux à chaque question. Si le nombre à trouver est 30, les intervalles après chaque question sont :

QUESTION	REPONSE	INTERVALLE
>,< ou =64	<64	1-63
>,< ou =32	<32	1-31
>,< ou =16	>16	17-31
>,< ou =24	>24	25-31
>,< ou =28	>28	
>,< ou =30	=30	trouvé!

6 questions (ou comparaisons) ont été nécessaires pour trouver le nombre. Avec la méthode précédente, il fallait 98 comparaisons. Le nombre maximum de comparaisons avec le tri binaire est pour n éléments de Ln 2 (ln=logarythme en base 2).

Le nom de la routine vient du fait que l'intervalle en question est divisé par deux à chaque fois. Pour cette raison également, on considère le logarythme en base 2. Les gains de temps sont énormes pour de longues listes.

La position d'un élément dans un ensemble de 60 millions d'éléments peut être trouvé en 26 comparaisons. Examinez avec attention le programme de tri binaire et simulez son déroulement à la main (faites l'ordinateur) afin de bien comprendre le fonctionnement de l'algorythme.

Compte tenu de la complexité de programme, le gain en rapidité ne se fait sentir qu'à partir d'ensembles qui ont plus de 100 éléments.

```
900 REM tri binaire
910 FOR i=1 TO anz
920 IF a(i)>a(i-1) THEN 1000
930 l=-1:r=i+1
940 h=INT((l+r)/2)
950 IF a(i)>a(h) THEN l=h ELSE r=h
960 IF r>l+1 THEN 940
970 a=a(i):FOR j=i TO r+1 STEP -1:a(j)=a(j-1):NEXT
980 a(r)=a
990 IF aus THEN GOSUB 170
1000 NEXT
1010 RETURN
```

TRI DE SHELL

L'algorythme du tri de shell tient son nom de son inventeur D.L. Shell, qui a découvert l'algorythme dans les années 50.

L'important dans cet algorythme est qu'il s'agit d'un tri à bulles, mais avec la différence que l'ensemble à trier est divisé en petits ensembles qui sont triés séparément ce qui petit à petit trie l'ensemble global.

L'idée originale est de trier l'ensemble en gros. Prenons pour exemple un ensemble avec 16 éléments. Pour le premier tri, on compare le premier et le 9ème élément, le 2ème et le 10ème, le 3ème et le 11ème, etc... et si nécessaire on les échange. Par ce moyen, on a effectué le premier tri grossier. Durant ce processus, les éléments distants de 8 ont été comparés.

Dans la seconde boucle, les éléments distants de 4 seront comparés. Ainsi, les sous-ensembles constitués des éléments séparés de 8 unités seront triés, c'est-à-dire le sous-ensemble constitué des 1er, 5ème, 9ème et 11ème éléments, celui constitué des 2ème, 6ème, 10ème et 14ème éléments, etc... Ainsi, on compare le 1er élément avec le 5ème et on les échange éventuellement, puis le 9ème comparé aux 1er et 5ème, et enfin le 13ème comparé aux 1er, 5ème et 9ème. L'opération est refaite avec les autres sous-ensembles. Puis l'intervalle est divisé par deux et on recommence.

Le tri de Shell est un algorythme très intéressant qui surpasse tous ceux dont il a été question jusqu'à maintenant.

```
1050 REM tri de shell
1060 FOR m=anz-1 TO 1 STEP -1
1070 m=INT((m+1)/2)
1080 FOR j=0 TO anz-m
1090 i=j
1100 IF a(i)<=a(i+m) THEN 1150
1110 s=a(i):a(i)=a(i+m):a(i+m)=s
1120 IF aus THEN GOSUB 170
1130 i=i-m
1140 IF i>=0 THEN 1100 ELSE 1150
1150 NEXT j,m
1160 RETURN
```

LE QUICKSORT

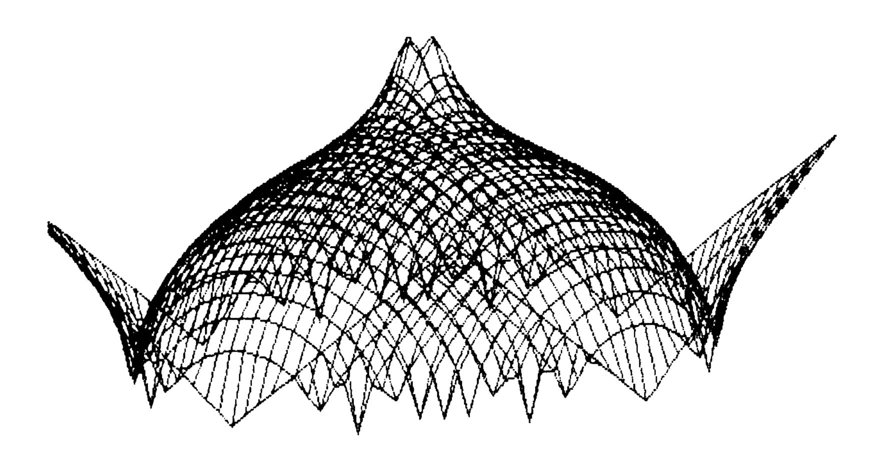
Un des moyen de tri les plus connus est le 'QUICKSORT'. Pour les ensembles qui contiennent plus de 50 éléments, il surpasse les algorythmes que nous avons présenté jusqu'ici. L'idée de base du QUICKSORT est la suivante :

L'ensemble à trier est divisé en deux parties égales. L'élément situé entre les deux parties est pris comme élément de comparaison. Puis tous les éléments de la première moitié qui sont plus grand que l'élément de comparaison sont changés de place. Dans certains cas, l'élément de comparaison lui-même est changé de place. Ce premier tri grossier est un tri de Shell amélioré.

Après la première itération, on recommence le processus avec chaque moitié. Le processus est récursif, c'est à dire qu'il s'appelle lui-même. Pour permettre cette structure récursive en BASIC, il faudra mémoriser les variables importantes avant la nouvelle éxécution, afin de ne pas les perdre. C'est le rôle des vecteurs l(sp) et r(sp).

```
1200 REM Quicksort
1210 sp=0:l(sp)=0:r(sp)=anz
1220 l=l(sp):r=r(sp):sp=sp-1
1230 i=l:j=r:vv=a(INT((l+r)/2))
1240 WHILE a(i)<vv :i=i+1:WEND
1250 WHILE a(j)>vv :j=j-1:WEND
1260 IF i>j THEN 1300
1270 s=a(i):a(i)=a(j):a(j)=s:IF aus THEN GOSUB 17C
1280 i=i+1:j=j-1
1290 IF i<=j THEN 1240
1300 IF i<r THEN sp=sp+1:l(sp)=i:r(sp)=r
1310 r=j:IF l<r THEN 1230
1320 IF sp>=0 THEN 1220
1330 RETURN
```

II. GRAPHISMES 3D



```
40 def fnf(x)=sqr(abs((16-x*x-z*z)))+1/sqr(x*x+z*z+.1)
50 ap=34
80 data -4,4,-2,6,-4,4
```

Une des possibilités les plus belles et les plus intéressantes du graphisme par ordinateur est la création de dessins tri-dimensionnels. La théorie complète du graphisme tri dimensionnel est très complexe et à pris de l'extension pendant les dernières années avec le développement des systèmes de CAO (conception assistée par ordinateur). Avec la dernière génération des ordinateurs, il est même possible de créer des scènes de films, par exemple de science fiction.

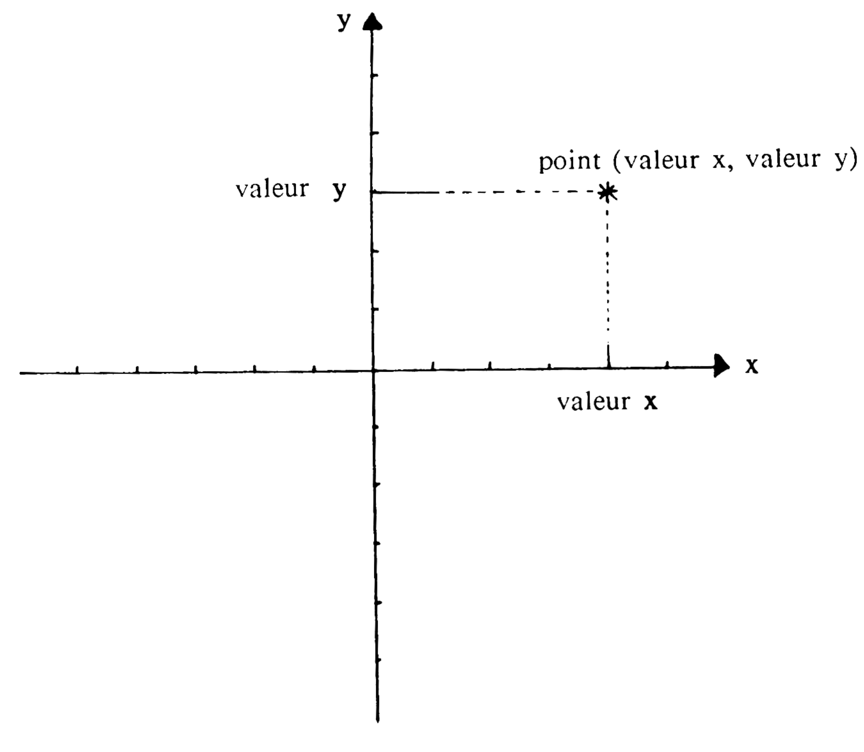
Naturellement, nous n'allons pas prendre cette voie. Mais le CPC propose une définition graphique encore inégalée dans sa classe. La résolution de 640*200 du mode 2 est idéale pour le dessin de fonctions tri dimensionnelles.

Voici maintenant un peu de théorie sur les fonctions.

Une fonction est une représentation. Une valeur d'un ensemble de départ (le domaine de définition) est affectée à une valeur d'un ensemble d'arrivée (l'ensemble des valeurs de la fonction), dans deux dimensions au moins. La relation entre ces deux valeurs est appelée fonction.

On peut donc définir une fonction comme un tableau, dans lequel chaque valeur du domaine de définition est reliée à une valeur de l'ensemble des valeurs de la fonction. La paire ainsi obtenue peut être représentée graphiquement, en dessinant deux axes portant chacune des valeurs et le point d'intersection entre les droites parallèles à ces axes passant par les points représentant les valeurs. Les deux valeurs en question sont appelées coordonnées.

Voici un exemple de représentation:



Chaque nombre réel valeur de cette fonction est obtenu en supprimant la partie décimale. Voici quelques valeurs de cette fonction:

X	Y
1,5	1
3,7593	3
1000,379	1000
2	2
-1,5	1

Nous allons dessiner cette fonction avec l'ordinateur.

Il faut d'abord la mettre sous forme mathématique de façon à ce qu'elle puisse être traitée par l'ordinateur. Dans notre cas, c'est très simple. Il s'agit de la fonction INT.

La variable des valeurs du domaine de définition est x. Celle pour l'ensemble des valeur de la fonction est y. La fonction doit être dessinée pour l'intervalle -5 à 15. Un premier jet pourrait être :

```
10 mode 2
20 for x=-5 to 15
30 y=int(x)
40 plot x,4
50 NEXT
```

L'effet désiré n'est pas obtenu. Nous devons encore adapter le programme au format de l'écran. La largeur de l'écran (direction x) est de 640 points et la hauteur de 200 points (direction y). La hauteur prend les valeurs 0 à 400 puis est divisée par deux par l'ordinateur.

L'image obtenue précédemment est une représentation miniature du dessin désiré. Elle doit encore être adaptée dans les directions x et y.

Nous allons définir deux facteurs d'agrandissement qui adapteront les valeurs de x et y à l'écran tout entier.
La fonction doit être représentée dans l'intervalle xobg=15 à xuntg=-5. L'ensemble à représenter a donc pour taille xobg-xuntg=15-(-5)=20. Comme nous avons 640 points, il faut adapter le 20 à 640, c'est à dire multiplier la valeur x par 640/20=32.

```
10 mode 2
20 xuntg=-5:xobg=15
30 masx=640/(xobg-xuntg)
40 for x=xuntg to xobg step 1/masx
50 y=int(x)
60 plot x*masx,y
70 next x
```

L'instruction step permet de dessiner un point pour chacune des 640 valeurs.

Le dessin obtenu est encore un peu plat car nous n'avons pas encore adapté la valeur y.

Comme pour l'axe des x, il faut savoir entre quelles limites supérieure et inférieure il faut dessiner la fonction. Cela n'est pas évident pour une nouvelle fonction et l'on doit soit faire des essais, soit étudier la fonction. On prend alors les valeurs obtenues comme limites et on trace la fonction.

Etant donnée la spécificité de notre fonction, il est facile de se rendre compte que y prendra les valeurs de -5 à 15. Ajoutez donc les lignes suivante ou changez les anciennes.

```
21 yuntg=-5:yobg=15
31 masy=400/(yobg-yuntg)
```

Il faut changer la ligne 60 en plot x*masx,y*masy

Malgré l'introduction du facteur d'adaptation mas, nous n'avons pas encore réussi à adapter la fonction à tout l'écran. Il est important de faire correspondre l'origine avec les limites supérieures ou inférieures de x et y. Normalement, l'origine est située au coin gauche en bas de l'écran.

En principe, nous pouvons adapter cette origine avec l'instruction ORIGIN. Dans notre cas, cela est rendu possible par la ligne :

```
35 origin 160,100
```

Mais, cette solution qui est la plus simple ne fonctionne que si le point d'origine est situé assez proche de l'écran réel. Mais si nous voulons par exemple dessiner la fonction dans l'intervalle 2000 à 2040, un ORIGIN -32000,20000 produirait une erreur overflow. Les calculs correspondant doivent être fait par programme. Nous devons déplacer l'écran de xuntg dans la direction x et de yuntg dans la direction y. L'adaptation de la ligne 60 est la suivante (effacez la ligne 35 et exécutez ORIGIN 0,0) :

```
60 plot (x-xuntg)*masx,(y-yuntg)*masy
```

Si nous ajoutons les lignes suivantes, nous voyons apparaître les axes de coordonnées sur l'écran, à condition qu'ils s'y trouvent réellement.

```
35 if xuntg*xobg>0 then 37
36 move (x-xuntg)*masx,0:draw (x-xuntg)*masx,400
37 if yuntg*yobg>0 then 40
38 move 0,(y-yuntg)*masy:draw 640,(y-yuntg)*masy
```

La fonction INT(x) décrite précédemment à une caractéristique inesthétique: en mathématique, on dit qu'elle n'est pas continue. Cela signifie que le graphe est tracé par escaliers. Par la suite, nous allons étudier uniquement des fonctions continues, car elles possèdent des originalité très intéressantes à programmer. Une des fonctions les plus simples est celle qui trace x en fonction de lui-même. Il est facile de le tester, le graphe de cette fonction est une droite diagonale qui coupe les axes à l'origine.

Nous prendrons pour exemple d'une fonction continue, un polynôme du second degré dont la forme la plus simple est: y=x^2. Définissons d'abord la fonction par DEF FN:

```
15 DEF FNY(x)=x^2
```

adaptation :

```
50 Y=FNY(x)
```

Etant donné qu'une fonction continue demeure semblable sur une part suffisamment courte de l'axe des x, il est possible de la dessiner au point par point. Ou bien on peut tracer des points à des coordonnées précises et les relier par des droites. Suivant la place des points, le gain en temps de tracé va de 10 à 50 fois. Le programme de tracé en trois dimensions qui se trouve à la fin du chapitre tient de ce principe. Il faut adapter le programme en conséquence :

```
16 puab=10: rem emplacement du point
```

Changer le numéro de ligne 40 en 42.

```
40 move 0,(fny(xuntg)-yuntg)*masy
42 for x=xuntg to xobg step 1/masx*puab
```

Transformation de la ligne 60:

```
draw (x-xuntg)*masx,(y-yuntg)+masy
```

En outre il faut adapter les limites de y à-10 et 250.
Essayez aussi les polynômes du second degré suivants:

fonction	xuntg	xobg	yuntg	yobg
x^2-50	-10	15	-150	200
x^2+20	-5	10	-10	90
(x-5)^2	-5	15	-10	100
(x+5)^2	-10	10	-10	250
(x-5)^2-50	-5	15	-60	60

Comme nous avons pu le constater, la représentation graphique de fonctions en deux dimensions est relativement simple. Elle se fait sur un plan et elle est donc facile à faire sur l'écran.
Mais pour les fonctions en trois dimensions, il n'est pas possible de faire une représentation directe car l'écran n'a que deux dimensions.

Dans une représentation tri dimensionnelle, chaque paire de valeurs de l'ensemble de définition est en relation avec une valeur de fonction. On peut par exemple imaginer que le domaine de définition d'une table est représenté par l'ensemble des points du plateau. Pour obtenir une représentation graphique réelle de la fonction, il faudrait modeler la forme qu'elle prend dans l'espace.

Le travail du programmeur est de représenter cette fonction tri dimensionnelle sur un plan. Une représentation fidèle doit tenir compte du fait que les parties les plus éloignées doivent être représentées en plus petit et les parties proches en plus grand. Avec cet effet, la perspective serait réelle.

Nous ne tiendrons pas compte de cet effet pour notre objectif qui est simplement la représentation graphique en trois dimensions d'une fonction. Imaginez une représentation spatiale de la fonction faite en pâte à modeler. Vous tracez des lignes parallèles sur la surface modelée. Nous avons déjà représenté la première ligne avec le vieux programme. Il faut maintenant tracer le reste.

Regardons cette fonction:

```
DEF FN(x)=x^2+z^2
```

Où z est la valeur portée sur le troisième axe. Dans notre exemple, elle serait le coté de la table.

Imaginons que la seconde courbe coupe l'axe de z en -1, c'est à dire qu'elle a une unité de moins que la première ligne. Cette ligne sera dessinée pour z ayant pour valeur -1 et en utilisant la routine de dessin. Puis on recommence avec la valeur de la courbe suivante, etc...

D'abord renumérotez la ligne 40 en 41. La ligne 15 contiendra la fonction définie précédemment.

```
22 zuntg=-10:zobg=0
40 for z=zobg to zuntg step-1
```

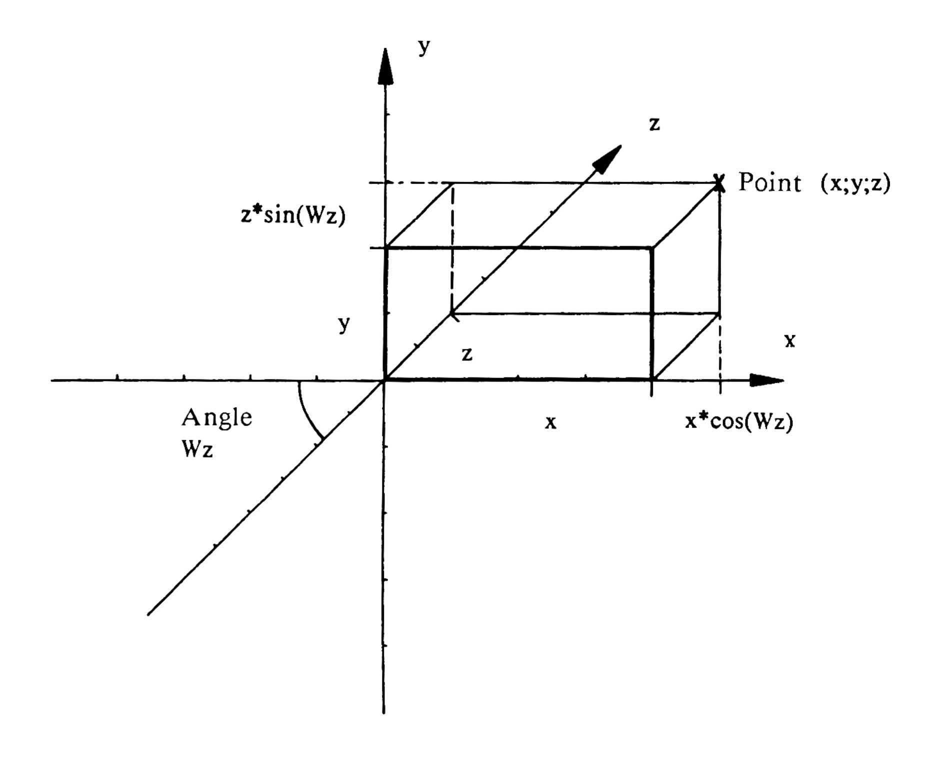
Transformez la ligne 41 en:

```
41 move 0,(fny(xuntg)-yuntg)*masy-z*masz*mazy
70 next
```

Le dessin obtenu ainsi correspond dans notre modèle à une vue de haut de la table. Choisissons un point de vue plus intéressant.

Dans la perspective standard, nous observons notre modèle de la droite. Dans cette représentation, l'axe des z descend vers la droite presque en diagonale. Un point qui serait situé loin sur l'axe z serait représenté sur l'écran vers le coin droit en haut.

Cela signifie que les points éloignés réellement sont représentés vers le haut pour l'axe des y et vers la droite pour l'axe des x. Plus le point est éloigné sur l'axe des z, plus le déplacement sur les coordonnées de l'écran est important.



Dans le programme, nous obtenons cela ainsi:

D'abord il faut changer la ligne 41 en:

```
41move-z*masz*mazx,(fny(xuntg)-yuntg)*masy-z*masz*masy
```

```
6 deg
25 wz=45:'angle de l'axe z avec l'axe x
26 mazx=cos(wz):masy=sin(wz)
32 masz=200/(zobg-zuntg)
60 draw (x-xuntg)*masx-z*masz*masx,(y-yuntg)*masy-z*masz*mazy
17 linab=10
```

Transformation:

```
40 for z=zobg to zuntg step 1/masz*linab
```

Changeons quelques lignes pour une nouvelle fonction et renumérotons le programme. Nous obtenons alors :

```
10 DEG
20 MODE 2
30 DEF FNy(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
40 puab=20: REM situation du point
50 linab=20
60 xuntg=-2:xobg=2
70 yuntg=-7:yobg=7
80 zuntg=-300:zobg=300
90 wz=45:'angle entre l'axe des Z et l'axe des X
100 mazx=COS(wz):mazy=SIN(wz)
110 masx=640/(xobg-xuntg)
120 masy=400/(yobg-yuntg)
130 masz=400/(zobg-zuntg)
140 IF xuntg*xobg>0 THEN 160
150 MOVE (-xuntg)*masx,0:DRAW (-xuntg)*masx,400
160 IF yuntg*yobg>0 THEN 180
170 MOVE 0,(y-yuntg)*masy:DRAW 640,(y-yuntg)*masy
180 FOR z=zobg TO zuntg STEP-1/masz*linab
190 MOVE -z*masz*mazx,(FNy(xuntg)-yuntg)*masy-z*masz*mazy
200 FOR x=xuntg TO xobg STEP 1/masx*puab
210 y=FNy(x)
220 DRAW (x-xuntg)*masx-z*masz*mazx,(y-yuntg)*masy-z*masz*ma
zy
230 NEXT x,z
```

Ce programme crée des graphismes tri dimensionnels en fil de fer. Souvent, cette méthode ne donne pas de beaux résultats. Représentez vous une fois de plus notre modèle en p*te à modeler et tracez des lignes perpendiculaires à celles déjà existantes. La surface du modèle est alors striée de lignes qui forment un filet.

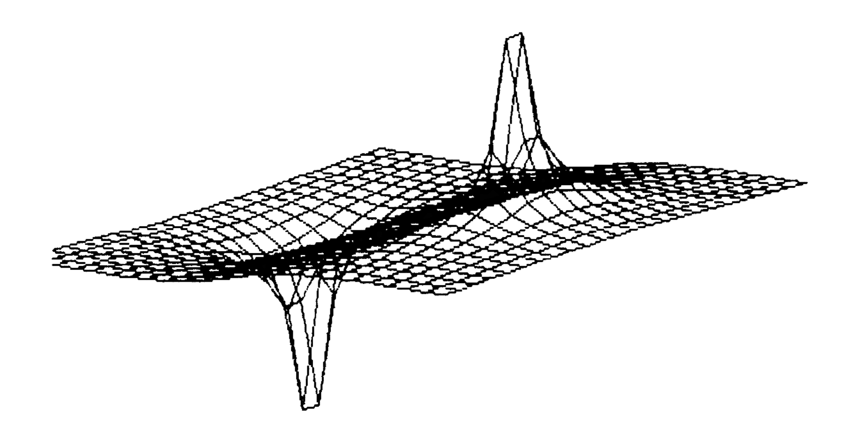
Pour obtenir ce résultat en deux dimensions, il suffit de reprendre le programme en changeant la boucle z avec la boucle x, testez le !

Dans ce cas, bien sûr, chaque point du filet sera calçulé deux fois, ce qui fait perdre du temps. Le programme qui suit permet de faire les deux boucles à la fois. Les points qui viennent d'être dessinés sont mémorisés et sont reliés avec les points correspondant de la prochaine ligne à tracer. Puis le processus est repris. Le filet complet est tracé de cette manière. Voici maintenant le programme et quelques images réalisées avec.

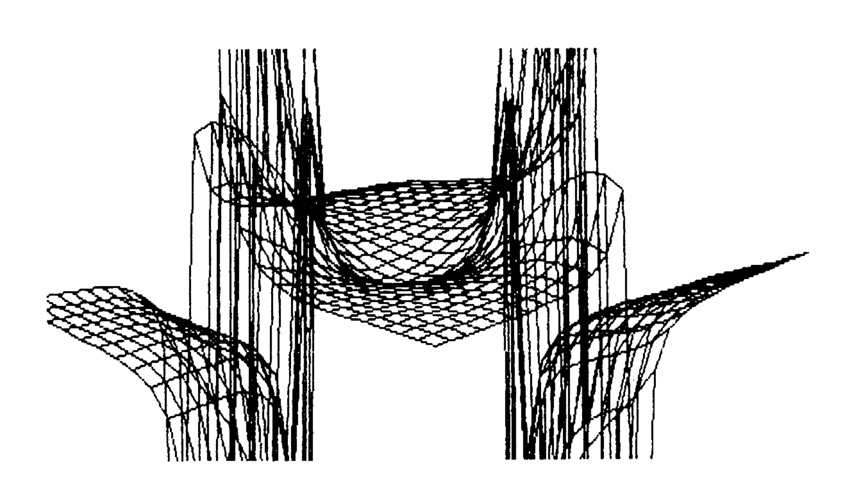
LISTE DES VARIABLES:

as:	angle entre l'horizontale et l'axe des x
al:	angle entre l'horizontale et l'axe des y
ap:	nombre de points sur la ligne du graphe
i:	compteur
j:	compteur
mz:	facteur d'agrandissement en z
my:	facteur d'agrandissement en y
mx:	facteur d'agrandissement en x
xk:	coordonnées x de l'écran
xy:	facteur de projection de x sur l'axe des y
xx:	facteur de projection de x sur l'axe des x
xe:	fin de x
xa:	début de x
yk:	coordonnées y
ye:	fin de y
ya:	début de y
y:	valeur de la fonction en y
zy:	facteur de projection z sur l'axe des y
zx:	facteur de projection z sur l'axe des x
ze:	fin de z
za:	début de z
z:	valeur z

```
10 MEMORY &9FFF
20 MODE 2
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=20
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-7,7,-300,300
90 mx=400/(xe-xa)
100 my=400/(ye-ya)
110 mz=400/(ze-za)
120 DEG
130 al=42-22/2
140 as=22
150 zx=COS(al)^2
160 xx=COS(as)^2
170 zy=SIN(al)^2
180 xy=SIN(as)^2
190 j=0
200 FOR z=ze TO za STEP -(ze-za)/ap
210 i=0
220 FOR x=xe TO xa STEP -(xe-xa)/ap
230 y=FNf(x)
240 xk=120+xx*mx*(x-xa)-zx*mz*(z)
250 yk=my*(y-ya)-zy*mz*(z)-xy*mx*(x-xa)
260 IF i=0 THEN 280
270 MOVE xk,yk:DRAW x(i),y(i)
280 i=i+1
290 IF j=0 THEN 310
300 MOVE xk,yk:DRAW x(i),y(i)
310 x(i)=xk
320 y(i)=yk
330 NEXT x
340 j=j+1
350 NEXT z
360 IF INKEY$="" THEN 360
```



```
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=23
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-9,7,-300,300
```



```
40 DEF FNf(x)=COS(4*x*x+z*z)*((x*x+z*z)/(x*z+1.1))+ABS(SIN(1
0*SQR(x*x)))
50 ap=24
80 DATA -4,4,-12,12,-4,4
```

III. PROGRAMMATION EN BASIC FACILITEE

3.1 Générateur de menus

Dans beaucoup de grands programmes, l'utilisateur peut faire son choix grâce à un menu. Dernièrement s'est marquée une tendance à rendre l'utilisation des programmes plus facile. Cette tendance se maintiendra sans doute, voire sera amplifiée. Un menu confortable à utiliser représente un gain considérable pour la facilité d'utilisation. Grâce au menu, l'utilisateur peut comme au restaurant choisir à la carte ce qu'il désire. Dans notre cas, ce sera une fonction du programme. Comme le choix sur un menu peut conduire à de très nombreuses possibilités et prend beaucoup de temps, les bons menus sont clairs et divisés en sous ensembles.

Un menu principal conduit à un sous-menu, qui lui même peut être éventuellement divisé jusqu'à ce que l'on arrive au point désiré. De plus un bon menu doit permettre d'avoir accès à certaines fonctions directement, ainsi qu'au premier ou au dernier menu.

Comme la programmation d'un menu est toujours utile, nous avons écrit un générateur de menu qui permettra un gain de temps appréciable.

Avec quelques adaptations minimes, en l'occurence la transformation de lignes de datas, le programme peut être utilisé universellement.

Naturellement, le programme est sans intérêt, sans le programme principal qui est appelé par le menu. Le programme principal est celui que vous avez préparé avec le menu. La technique du générateur de menu montre une utilisation intéressante des variables locales du BASIC. Sans cela, il eut été presque impossible de le réaliser. L'objectif était de réaliser chaque menu avec une seule routine. Une fois que l'utilisateur à fait son choix, la routine doit trouver le chemin dans le nouveau menu.

Tous les points du menu sont mémorisés dans une seule variable (m$) l'un après l'autre. Pour garder les différents points du menu dans l'ordre nécessaire, il existe une seconde variable (md pour menu-data), dans laquelle les informations suivantes sont mémorisées pour chaque menu :

- numéro du menu suivant
- numéro du dernier menu
- taille du menu (nombre de points)
- le choix de la fonction

La variable mepo contient le numéro du menu courant. Avec m$(mepo), on obtient le nom de la partie de programme dans laquelle on se trouve.

md(mepo,gros) contient le nombre de choix du menu courant, md(mepo,dflt) contient le point du menu qui a été choisi en dernier.

Le numéro du menu auquel on peut avoir accès par la touche DEL (donc le dernier menu) est mémorisé dans md(mepo,ruck).

Enfin, le numéro du menu suivant (accessible par le premier choix) est mémorisé dans md(mepo,nxt). Etant donnée la structure de données complexe, md(mepo,nxt) possède encore quelques autres fonctions. D'une part, on peut trouver le numéro du menu suivant et d'autre part, cette valeur est celle du premier choix du menu courant (voir lignes 520-560).

La partie la plus compliquée du programme est la boucle qui défini le pointeur décrit précédemment à partir des lignes de data dans lesquelles se trouvent les choix du menu. Les lignes de data doivent donc contenir les choix du menu avec lesquels chaque menu est appelé par un autre (avec un nombre). Avec 0, le menu principal est appelé puis suivent les sous-menus qui succèdent dans l'ordre correspondant. Les sous-menus sont crées de la même manière. Si il n'existe pas de sous-menu pour le choix effectué, il faut laisser un blanc dans la numérotation.

La boucle d'initialisation (lignes 10100 à 10130) lit d'abord les points du menu qui suivent. Pour chaque choix enregistré, le pointeur de branchement est fixé pour le choix effectué : md(i,ruck) prend la valeur du menu n courant. La valeur par défaut (dflt), celle du dernier choix effectué, est fixée à 1 : c'est à dire sur le premier choix possible. Puis la routine vérifie que tous les points ont été lus. A la fin de la ligne data, il faut inscrire un '!'.

Si tous les points ont été lus, la taille du dernier menu est déterminée et la boucle d'initialisation est terminée.

La ligne 10120 teste si un numéro a été lu. Si ce n'est pas le cas, le point suivant du menu est lu en ligne 10110. S'il apparaît qu'un numéro de menu a été lu, n prend la valeur de ce nouveau numéro. La procédure suivante peut paraître illogique.

Le compteur courant du menu qui vient d'être lu est sauvegardé en tant que numéro du menu suivant. (Testez si cette valeur est bien celle du menu suivant.) Puis on détermine la taille du dernier lu.

Après le fonctionnement de la boucle d'initialisation, apparaît l'image suivante :

Après l'initialisation du programme, elle peut être utilisée dans tous les programmes. La routine d'affichage du menu a le format suivant:

```
1010 gosub 500: if ein$=del$ then return
1020 on wahl gosub ...numéro de ligne
```

Dans les lignes appelées, se trouvent des ordres semblables qui, comme aux lignes 1010 et 1020, se branchent sur le menu suivant.

```
10 REM menu
20 MODE 2
30 GOSUB 1000 : REM Init
40 GOSUB 300 : REM menu principal
50 CLS:END
60 REM bordure
70 CLS:PRINT"generateur de menus
   ";m$(mepo)
80 PRINT STRING$(80,"-");
90 LOCATE 1,23:PRINT STRING$(80,"-");
100 PRINT"Appuyez sur  >ENTER< pour l'affichage en video inv
erse"
110 PRINT"          ou >DEL< pour ";exit$(-(mepo>0)-(mepo>1))
;
120 RETURN
130 REM Construction du menu
140 GOSUB 60 :REM bordure
150 choix=md(mepo,dflt)
160 LOCATE 1,5
170 FOR i=1 TO md(mepo,gros):IF i=choix THEN PAPER 1:PEN 0
180 PRINT i;:PAPER 0:PEN 1:PRINT" - ";:IF i=choix THEN PAPER
 1:PEN 0
190 PRINT m$(i+md(mepo,nxt)-1):PAPER 0:PEN 1
200 NEXT
210 PRINT:PRINT:PRINT:PRINT"votre choix :";:PAPER 1:PEN 0:PR
INT choix;:PAPER 0: PEN 1:LOCATE POS(#0)-3,VPOS(#0)
220 ein$=INKEY$:IF ein$="" THEN 220
230 IF ein$=del$ THEN RETURN
240 IF ein$=CHR$(13) THEN md(mepo,dflt)=choix:mepo=md(mepo,n
xt)+choix-1:RETURN
250 IF ein$=CHR$(241) OR ein$=CHR$(242) OR ein$=CHR$(32) THE
N choix=choix+1+(choix=md(mepo,gros))*md(mepo,gros):GOTO 290
260 IF ein$=CHR$(240) OR ein$=CHR$(243) THEN choix=-(choix=1
)*md(mepo,gros)+choix-1:GOTO 290
270 n=VAL(ein$):IF (n<1) OR (n>md(mepo,gros)) THEN 220
280 choix=n
290 PAPER 1:PEN 0:PRINT choix;:PAPER 0:PEN 1:GOTO 160
300 REM Menu principal
310 GOSUB 130:IF ein$=del$ THEN RETURN
```

```
320 ON choix GOSUB 340,3000,4000,5000,6000
330 GOTO 310
340 REM a cote 1
350 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck): RETURN
360 ON choix GOSUB 370,2200,2300,2400,2500,2600:GOTO 350
370 REM sous1
380 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck): RETURN
390 ON choix GOSUB 2150,2160,2170:RETURN
1000 REM Init
1010 del$=CHR$(127)
1020 FOR i=0 TO 2:READ exit$(i):NEXT
1030 DATA "fin du programme"
1040 DATA "menu principal"
1050 DATA "dernier menu"
1060 DIM m$(100),md(100,3):ruck=0:nxt=1:gros=2:dflt=3:n=-1:i
=-1
1070 i=i+1:READ m$(i):md(i,ruck)=n:md(i,dflt)=1:IF m$(i)="!"
 THEN i=i-1:md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+1:RETUR
N
1080 IF LEN(m$(i))<3 THEN n=VAL(m$(i)):md(n,nxt)=i:i=i-1:IF
md(i,ruck)>=0 THEN md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+
1
1090 GOTO 1070
1100 RETURN
1110 DATA "menu principal"
1120 DATA 0
1130 DATA "impression"
1140 DATA "affichage"
1150 DATA "edition"
1160 DATA "recherche"
1170 DATA "aides"
1180 DATA 1
1190 DATA "Normal","Proportionnel","Blocs","Blocs+Proportion
nel"
1200 DATA 6
1210 DATA "dernier menu","seulement pour des tests","etc. et
c. etc."
1220 DATA "!"
```

3.2. Générateur de masque de saisie

Dans un programme facile à utiliser, il est nécessaire de prévenir toutes les manipulations de l'utilisateur contre les mauvaises manoeuvres.

Les interventions de l'utilisateur se font sur toutes les parties du programme où l'utilisateur peut modifier le déroulement du programme avec le clavier, la manette de jeu, le crayon lumineux ou tout autre outil de saisie. Le générateur de menu du chapitre présenté précédemment est écrit de telle manière que tous les choix erronés sont ignorés. Dans un masque de saisie, cela doit être réalisé pour n'importe quel choix désiré. Prenons un exemple :

Il faut saisir par exemple, pour un programme de gestion d'adresse, le nom, l'adresse, le lieu, le numéro de téléphone et la date. Pour chaque saisie, il faut fixer une fenêtre définie. Lors du chargement du nom, seules certaines lettres sont autorisées, lors du chargement du numéro de téléphone, ce sont certains chiffres et le tiret, etc...

Les données pour la saisie doivent être saisie par les lignes de data. Les informations suivantes sont importantes pour cela :

- Commentaire pour la saisie (p.ex. nom)
- La position du commentaire à l'écran.
- Longueur de la saisie
- Chiffre indiquant le nombre des saisies possibles.

La saisie d'un masque avec tous ses secteurs est finie lorsque l'on se trouve dans un nouveau secteur de saisie, à condition bien entendu qu'il ne soit pas possible de corriger la saisie précédente. Une saisie plus longue que celle autorisée ne doit pas être possible. Les touches du curseur pour gauche et droite déplacent le curseur dans les secteurs. RETURN et la touche 'curseur vers le bas' déplacent le curseur dans le secteur suivant et la touche 'curseur vers le haut' ramène le curseur dans le secteur précédent. Avec la touche COPY, la saisie est reprise du début.

```
10 MODE 2
20 GOSUB 820
30 GOSUB 80: REM saisie des donnees
40 REM donnees disponibles
50 LOCATE 1,20
60 FOR i=1 TO ensemble:PRINT ein$(i):NEXT
70 END
80 CLS:ensemblenum=1
90 FOR i=1 TO ensemble
100 LOCATE x(i),y(i)
110 PRINT coment$(i);":";STRING$(l(i),"."):ein$=STRING$(l(i)
," ")
120 ein$(i)=STRING$(l(i)," ")
130 NEXT
140 x=x(ensemblenum):y=y(ensemblenum)
150 lm=l(ensemblenum):kenzif=ken(ensemblenum)
160 LOCATE x,y:PRINT coment$(ensemblenum);":";
170 x=x+LEN(coment$(ensemblenum))+1
180 IF x>80 THEN y=y+x\80:x=x MOD 80
190 e$=ein$(ensemblenum)
200 GOSUB 280
210 ein$(ensemblenum)=e$
220 IF endflg THEN RETURN: REM saisie terminee
230 IF hochflg THEN ensemblenum=ensemblenum-1:IF ensemblenum
=0 THEN ensemblenum=1
240 IF runtflg THEN ensemblenum=ensemblenum+1:IF ensemblenum
>ensemble THEN ensemblenum=ensemble
250 runtflg=0:hochflg=0
260 GOTO 140
270 PRINT CHR$(7);:GOTO 360
280 ' saisie des donnees
290 ' x,y : coordonnees de depart
300 ' lm : longueur max du champ de saisie
310 ' aus : Window #
320 ' chaine retournee : e$
330 LOCATE x,y
340 l=1
350 CALL &BB81: REM Curseur on
360 a$=INKEY$:IF a$="" THEN 360
```

```
370 a%=ASC(a$)
380 FOR i=1 TO anstz:IF a%<>stz%(i) THEN NEXT
390 ON i GOTO 470,470,510,540,570,590
400 REM Test de validite de la saisie
410 okflg=-1
420 ON kenzif GOSUB 610,660,720,770
430 IF okflg=0  THEN 270
440 IF POS(#aus)>=x+lm THEN 270
450 e$=LEFT$(e$,POS(#aus)-x)+a$+RIGHT$(e$,lm-(POS(#aus)-x)-1
)
460 PRINT a$;:GOTO 360
470 REM return et curseur vers le bas
480 runtflg=-1
490 CALL &BB84: REM Curseur off
500 RETURN
510 REM chr$(242) curseur a droite
520 IF POS(#aus)=x  THEN 270
530 PRINT CHR$(8);:GOTO 360
540 REM chr$(243) cursor links
550 IF POS(#aus)>=x+lm THEN 270
560 PRINT CHR$(9);:GOTO 360
570 REM CURSEUR vers le haut
580 hochflg=-1:GOTO 490
590 REM Copy
600 endflg=-1:GOTO 490
610 REM Test pour autoriser les lettres
620 IF a%>64 AND a%<91 THEN RETURN
630 IF a%>96 AND a%<123 THEN RETURN
640 IF a%=32 THEN RETURN
650 okflg=0:RETURN
660 REM Test pour autoriser les lettres et les chiffres
670 GOSUB 610
680 IF okflg THEN RETURN ELSE okflg=-1
690 REM Test des chiffres
700 IF a%>47 AND a%<59 THEN RETURN
710 okflg=0:RETURN
720 REM Test pour autoriser les n0 de tel, c-a-d chiffres et
 "/"
730 GOSUB 690
740 IF okflg THEN RETURN ELSE okflg=-1
```

```
750 IF a%=47 THEN RETURN
760 okflg=0:RETURN
770 REM Test pour autoriser les dates, c-a-d les chiffres et
 les points
780 GOSUB 690
790 IF okflg THEN RETURN ELSE okflg=-1
800 IF a%=46 THEN RETURN
810 okflg=0:RETURN
820 REM init
830 ' anstz : numero du caractere de controle
840 ' stz%(anstz) : ASCII du caractere de controle
850 anstz=6:FOR i=1 TO anstz:READ stz%(i):NEXT
860 DATA 13,241,242,243,240,224
870 aus=0:REM numero de fenetre
880 REM masque de saisie
890 READ ensemble : REM nombre de champs de saisie
900 FOR i=1 TO ensemble
910 READ coment$(i),x(i),y(i): REM commentaire, position x e
t y
920 READ l(i): REM longueur du champ de saisie
930 READ ken(i):REM caractere de controle pour les saisies a
utorisees
940 ' 1:lettre, 2:lettres et chiffres, 3:Telephone, 4:Date
950 NEXT i
960 RETURN
970 REM donnees du menu
980 DATA 5
990 DATA "Nom     ",1,4,20,1
1000 DATA "Date ",50,4,8,4
1010 DATA "rue      ",1,6,20,2
1020 DATA "commune",1,9,30,2
1030 DATA telephone,50,9,12,3
```

Liste des cross-references avec XREF

```
AUS ! 440 450 450 520 550 870
ANSTZ ! 380 850 850
A % 370 380 620 620 630 630 640 700 700 750 800
A $ 360 360 370 450 460
COMENT $ 110 160 170 910
ENDFLG ! 220 600
E $ 190 210 450 450 450
ENSEMBLENUM ! 80 140 140 150 150 160 170 190 210 230 230 230
 230 240 240 240 240
EIN $ 60 110 120 190 210
ENSEMBLE ! 60 90 240 240 890 900
HOCHFLG ! 230 250 580
I ! 60 60 90 100 100 110 110 110 120 120 380 380 390 850 850
 900 910 910 910 920 930 950
KEN ! 150 930
KENZIF ! 150 420
LM ! 150 440 450 550
L ! 110 110 120 150 340 920
M $
OKFLG ! 410 430 650 680 680 710 740 740 760 790 790 810
RUNTFLG ! 240 250 480
STZ % 380 850
X ! 100 140 140 160 170 170 180 180 180 180 330 440 450 450
520 550 910
Y ! 100 140 140 160 180 180 330 910
```

IV. CERCLE

Le listing qui suit permet de simuler l'ordre CIRCLE à partir du BASIC et de façon rapide. Cette fonction n'est pas disponible sur les nouvelles versions du cpc.

Etant donné que la routine ne calcule qu'un quart du cercle et trace le reste avec un système de 'miroir', elle est beaucoup plus rapide que la routine de cercle normale.

Lorsque l'on appelle la routine, R doit contenir le rayon, X et Y contiendront la position et E l'aplatissement.

```
10 RAD
20 MODE 2
30 INPUT"rayon,aplatissement?",r,e
40 INPUT"coord.x, coord.y?",xk,yk
42 GOSUB 60
43 END
60 x(0)=0:y(0)=r*e
70 x(1)=0:y(1)=-r*e
80 x(2)=0:y(2)=-r*e
90 x(3)=0:y(3)=r*e
100 FOR al=0 TO PI/2 STEP PI/18
110 x=r*SIN(al):y=SQR(r*r-x*x)*e
120 i=0:GOSUB 180
130 i=1:y=-y:GOSUB 180
140 i=2:x=-x:GOSUB 180
150 i=3:y=-y:GOSUB 180
160 NEXT
170 RETURN
180 MOVE xk+x(i),yk+y(i):x(i)=x:y(i)=y
190 DRAW xk+x(i),yk+y(i):RETURN
```

V. PROTECTION DES PROGRAMMES PERSONNELS

On peut protéger facilement ses propres programmes en les sauvegardant avec le suffixe 'p'. Le 'p' est rattaché au nom du programme, séparé par une virgule.

```
SAVE 'nom.xxx',p
```

Un programme sauvegardé ainsi ne peut être utilisé qu'avec l'instruction RUN 'nom.xxx'. Mais il est encore possible d'arrêter le programme en cours avec la touche ESC. Il n'est pas possible d'accéder directement au listing, mais les bidouilleurs y arrivent très rapidement.

Si on donne la valeur &c9 à la case mémoire &bdee:

```
POKE &bdee,&c9
```

Il n'est plus possible d'arrêter le programme avec la touche ESC. On obtient le même résultat avec l'instruction KEY DEF en BASIC :

```
KEY DEF 66,1,0   Touche ESC inhibée
KEY DEF 66,1,252 Touche ESC activée
```

Il est évident que le programme doit alors être développé astucieusement pour ne pas s'arrêter ou bien conduire à une erreur due à une mauvaise saisie, ce qui suspendrait son fonctionnement. Si un programme est trop complexe pour que l'on puisse déterminer toutes les possibilités d'erreur, on peut éviter l'arrêt du programme avec l'instruction:

```
ON ERROR GOTO numéro de ligne
```

Si vous définissez un message d'erreur spécifique ou si vous executez un call 0 à la ligne donnée, l'utilisateur ne pourra pas avoir accès au programme.

Exemple:

```
10 compteur=0
20 input 'diviseur',t
30 print 10/t
40 on error goto 100
50 goto 20
100 compteur=compteur+1
110 if compteur<4 then print 'mauvaise saisie' else call 0
120 goto 50
```

Dans cet exemple, l'utilisateur a l'autorisation de se tromper trois fois. Si il fait une quatrième mauvaise saisie, le call 0 exécutera un RESET.

Mais tout bon programme doit s'arrêter à un moment ou un autre. Pour éviter que la fin du programme ne permette de le cracker, il faut l'effacer de la mémoire. C'est encore la meilleure protection contre le listing. Si l'utilisateur répond positivement à la question concernant l'arrêt du programme, il faut exécuter un call 0. Ceci effectue un reset, ce qui peut être obtenu également avec CTRL/SHIFT/ESC.

VI. COMMENT FACILITER LA SAISIE DE PROGRAMMES

Lorsque l'on saisi de longs programmes, il est fréquent de laisser des blancs inutiles. Ils occupent de la place mémoire qui pourrait être utile au programmeur. Il est difficile de se concentrer à la fois sur la syntaxe et sur les blancs utiles ou inutiles.

Il suffit de donner une valeur non nulle à la case mémoire &ac00 pour résoudre ce problème. Dans ce cas, l'ordinateur contrôle quels sont les blancs nécessaires à la syntaxe et supprime les autres. Bien sûr, on peut aussi taper des blancs supplémentaires 'au cas où'. A la fin, seul les blancs nécessaires seront pris en compte.

VII. ACCELERATION DES PROGRAMMES EN BASIC

Bien que le BASIC du CPC n'ait pas de défauts de rapidité, il y a des problèmes pour lesquels le facteur temps joue un rôle primordial (par exemple pour les tris). Lorsqu'il n'est plus possible d'accélérer le processus en améliorant l'algorythme, certaines techniques de programmation permettent de forts gains de temps.

Nous allons expliquer certaines d'entre elles.

La variable BASIC TIME est un bon moyen de mesure pour le temps de fonctionnement du BASIC. Revenons sur le tri à bulles. Avec cet exemple, nous allons montrer comment accélérer les programmes. Dans le programme qui suit, nous avons écrit une partie fixe et une partie variable pour rendre les exemples plus clairs. Les lignes 10 à 90 demeureront les mêmes.

```
10 REM rapidite du BASIC
20 anz=10:anz=anz-1
30 DIM element(anz)
40 FOR i=0 TO anz:READ element(i):NEXT
50 DATA 10,4,7,3,2,24,8,17,5,19
60 t!=TIME
70 GOSUB 100
80 PRINT (TIME-t!)/300
90 END
100 REM tri a bulles
110 taille=anz : REM  L'ensemble a trier est d'abord
 l'ensemble complet
120   indice=1 : REM  comparaison avec le premier element
130     IF element(indice-1)>element(indice) THEN GOSUB 190
: REM  echange
140     indice=indice+1 : REM  comparaison avec l'element
 suivant
150     IF indice<=taille THEN 130 : REM  pour comparer
160   taille=taille-1 : REM  l'ensemble a trier est reduit
 de un
170   IF taille>=1 THEN GOTO 120 : REM  tri de l'ensemble
 reduit
180 RETURN
190 REM sous-programme : Echange de deux elements
200 variable=element(indice)
210 element(indice)=element(indice-1)
220 element(indice-1)=variable
230 RETURN
```

Temps : .62 secondes avec les lignes rem

LIGNES DE REM

La première amélioration est obtenue en supprimant les rem et ''' (shift+7)

temps : .61 secondes sans lignes de rem

CHOIX DES VARIABLES

Un des points les plus importants est le choix des variables. Le BASIC comprend les variables INTeger et REAL. Les variables REAL tiennent sur 5 octets et demandent plus de temps de traitement que les variables INT qui tiennent sur 2 octets. Par conséquent, il est préférable d'utiliser aussi souvent que possible les variables INT. La plupart du temps, on peut se passer des variables REAL.

```
5 DEFINT a-z
100 taille=anz
120 indice=1
130 IF element(indice-1)>element(indice) THEN GOSUB 200
140 indice=indice+1
150 IF indice<=taille THEN 130
160 taille=taille-1
170 IF taille>=1 THEN GOTO 120
180 RETURN
200 variable=element(indice)
210 element(indice)=element(indice-1)
220 element(indice-1)=variable
230 RETURN
```

temps= .45 secondes sans variables REAL

Les variables INT ne peuvent prendre que des valeurs allant de -32768 à 32767. Pour manipuler des adresses allant de 0 à 65535 (&0000 à &ffff), cela pose des problèmes. Mais au lieu d'utiliser des variables REAL, on peut manipuler les adresses exédant &7fff comme des nombres négatifs.

BOUCLES FOR-NEXT

Les boucles FOR-NEXT sont plus rapides que les boucles faites avec IF THEN GOTO.

Temps: .34 secondes avec des boucles FOR-NEXT

NOM ET DEFINITION DES VARIABLES

Plus le nom d'une variable est court, moins le temps de traitement est long. Comme le type de la variable influe sur la longueur, toutes les variables devraient tout d'abord être définies avec DEF INT ou REAL ou STR.

```
5 DEFINT a-z
10 REM rapidite du BASIC
20 a=10:a=a-1
30 DIM e(a)
40 FOR i=0 TO a:READ e(i):NEXT
100 FOR f=a TO 1 STEP -1
110 FOR i=1 TO f
120 IF e(i-1)>e(i) THEN GOSUB 150
130 NEXT i,f
140 RETURN
150 z=e(i)
160 e(i)=e(i-1)
170 e(i-1)=z
180 RETURN
```

temps: .35 secondes avec des noms de variables plus courts

LIGNES DE BASIC

Le traitement d'une ligne de BASIC longue est toujours plus court que le traitement de plusieurs petites. Les lignes non utilisées en tant que lignes de branchement devrait être groupées avec les autres.

```
100 FOR f=a TO 1 STEP -1:FOR i=1 TO f:
IF e(i-1)>e(i) THEN GOSUB 120
110 NEXT i,f:RETURN
120 z=e(i):e(i)=e(i-1):e(i-1)=z:RETURN
```

temps : .34 secondes avec moins de lignes

Comme vous pouvez le constater, nous avons gagné un temps considérable par rapport à la première version. Si vous ne trouvez pas cela encore assez rapide, vous devrez vous mettre de gré ou de force au langage machine.

Encore quelques trucs qui n'ont pas pu être démontrés avec l'exemple précédent.

- Avant d'utiliser des variables dont le nom commence avec les mêmes lettres, il vaut mieux utiliser les lettres de commencement qui n'ont pas encore été utilisées.

- Si vous devez utiliser des variables commençant par les mêmes lettres, il faudra d'abord initialiser les variables les plus utilisées, même lorsque cela n'est pas nécessaire. Exemple :

I est plus souvent utilisées que IN$. IN$ sert à la ligne 10 et I à la ligne 100 en premier. Il faut donc avant la ligne 10 écrire une ligne 9 I=0, par exemple.

Les variables trouvées en premier sont inscrites dans le tableau de variables en premier et sont donc retrouvées plus rapidement par la suite.

- Pour les calculs, les opérations de base sont toujours plus rapides. Ainsi, il vaut mieux employer x*x plutôt que x^2. De même, il faut éviter les parenthèses quand elles ne sont pas nécessaires.

- N'effectuez pas d'opération inutile dans une boucle. Par exemple, si une couleur spéciale doit être choisie pour une boucle, cela peut être réalisé avant la boucle.

Mais même les instructions qui ne conduisent à aucune opération, comme rem ou data, doivent être hors des boucles. Elles ne sont pas exécutées mais elles sont interprétées à chaque fois, ce qui est gourmand en temps.

- Evitez les blancs inutiles. Ils seront interprétés eux aussi et consomment du temps.

- Si vous travaillez beaucoup avec les chaînes, cela peut conduire à un 'dépôt d'ordures'. Toutes les valeurs inutilisées doivent être effacées de la mémoire et libérer de la place. Cette action dure souvent plusieurs secondes durant lesquelles l'ordinateur est inutilisable. Mais il ne faut pas l'éviter pour autant. Il faut la mener en plusieurs fois, afin de n'utiliser que quelques millisecondes à chaque fois. L'instruction PRINT FRE("") efface le 'dépôt d'ordures'. Si on le place à un endroit judicieux, par exemple après un INPUT qui prend beaucoup de temps, ou avec l'ordre EVERY-GOSUB, qui l'appelle à intervalles donnés, on peut éviter de perdre trop de temps avec cette opération.

VIII. INSTRUCTIONS SPECIALES DU BASIC DU CPC

8.1. EVERY a,b GOSUB c

Avec l'instruction EVERY-GOSUB, on a la possibilité fantastique de pouvoir exécuter des interruptions machine à partir du BASIC. Cela permet au programmeur de faire du multitâche en BASIC sans avoir à écrire des routines complexes en langage machine.

Avec cette possibilité, l'ordinateur peut exécuter plusieurs programmes simultanément. En outre, on peut gérer des sous programmes qui ont besoin d'une bonne gestion du temps, comme par exemple un timer.

Si l'instruction EVERY est appelée avec les paramètres que nous allons expliquer, l'interpréteur BASIC arrète toute activité pour se brancher sur la partie du programme appelée par le GOSUB.

Avant de développer les explications, voici un petit programme qui montrera l'utilité de cette instruction puissante.

```
5 mode 2: temps=0
10 every 50,0 gosub 100
20 for i=0 to 2*pi step pi/360
30 plot 320+300*sin(i),200+195*cos(i)
40 next i
50 end
100 temps=temps+1
110 locate 50,10
120 print temps
130 return
```

Cet exemple qui dessine un lissajou est courant mais il démontre bien l'utilité de l'ordre EVERY.

EVERY appelle un sous programme qui incrémente le compteur de temps et qui l'affiche. Naturellement, le sous programme doit être appelé au bon moment. Mais comme le temps de calcul des fonctions sin et cos est dépendant de la valeur de la fonction dans certaines limites (ainsi le calcul de sin pi/3 dure plus longtemps que le calcul de sin pi/4), on peut se passer de définir le temps de cette instruction plot. Cela est réalisé dans le sous programme qui est appelé à intervalles réguliers par l'instruction EVERY.

Les paramètres a,b et c:

a- indique le nombre d'unités de temps entre chaque appel de sous programme (unité=.02 sec).

b- est le numéro du compteur de temps. On dispose de quatre compteurs de temps avec les numéros 0 à 3 et 3 a la plus forte priorité. S'il y a plusieurs appels de sous programmes, ils sont gérés selon leur rang. Si par exemple, on a un appel avec la priorité 2, alors que le programme exécute une interruption avec un plus faible rang de priorité, celle ci est arrêtée, puis celle de plus haute priorité est exécuté et enfin la précédente est reprise. Il faut donc prévoir avec précision la distribution des priorités.

c- indique le numéro de ligne du sous programme. Le système d'exploitation se branche sur cette ligne à chaque intervalle précisé par a.

Les timers de plus forte priorité passent donc avant ceux de plus faible priorité. Tous les timers ont priorité sur le programme principal. Cela peut conduire au fait qu'une programmation maladroite ne laisse plus de temps pour le programme principal car les opérations réalisées utilisent trop de temps. Le programme de la page suivante en est un exemple.

```
5 mode 2
10 every 30 gosub 100
20 for i=1 to 639: plot i,100:next i
50 end
100 rem sous programme
110 for j=1 to 30: locate 20,5:print j:next j
120 return
```

Le sous programme en ligne 100 prend tellement de temps que le programme qui trace une ligne dispose à peine du temps suffisant. L'instruction every ne doit pas être utilisée de cette façon.

Pour bien maitriser l'instruction EVERY, lisez les explications sur les instructions EI et DI qui sont décrites dans le chapitre traitant de l'ordre AFTER.

8.2. AFTER a,b GOSUB c

L'ordre AFTER exécute un saut à un sous programme. Le branchement n'est effectué qu'après un temps déterminé.

Les paramètres :

a- nombre d'unités de temps (0,02 sec) après lesquelles le branchement doit être exécuté.

b- Numéro du timer. Il y a quatre timer désignés par 0 à 3, 3 ayant la plus haute priorité.

c- Donne la ligne où débute le sous programme.

Contrairement à l'instruction EVERY, qui conduit à des branchements successifs à un sous programme, l'instruction AFTER n'est suivie que d'un seul branchement.

Les instructions AFTER-GOSUB avec les plus haute priorités interrompent l'exécution des sous programmes appelés par des AFTER-GOSUB de plus faible priorité. Il seront exécutés plus tard.

Les instructions EVERY et AFTER simulent par logiciel des interruptions qui sont exécutées de façon électronique par le Z80. Par conséquent le BASIC du CPC comprend les instructions EI et DI du Z80.

DI signifie DISABLE INTERRUPT (inhiber l'interruption).
EI signifie ENABLE INTERRUPT (autoriser l'interruption).

Si un sous programme qui a été appelé par EVERY ou AFTER doit être absolument protégé contre les interruptions de plus forte priorité, on peut l'éviter par DI. Cette instruction inhibe les interruptions qui peuvent intervenir, qu'elles soient effectuées par EVERY ou AFTER.

De même, DI protège un sous programme appelé par exemple par EVERY 15,2 GOSUB, contre une coupure occasionnée par AFTER 10,2 GOSUB. Il faut remarquer dans cet exemple que le même timer a été précisé.

Avec l'instruction EI, toutes les interruptions sont de nouveau activées. Les interruptions appelées jusqu'alors n'ont pas été exécutées. Elles ont été mémorisées et peuvent être utilisées après EI rapidement.

Exécutez le programme suivant:

```
10 every 20,0 gosub 100
20 every 10,2 gosub 200
30 goto 30
90 rem——————————
100 di
110 for i=1 to 10:print i;:next
120 print :ei: return
130 rem——————————
200 print 'interruption priorité 2'
210 return
```

Changez maintenant la valeur de fin de boucle FOR-NEXT en lignes 10 et 50. On voit que la boucle, protégée par di est exécutée entièrement avant que l'interruption de priorité 2 n'intervienne en ligne 200. Mais alors, le texte de la ligne 200 est affiché non une mais autant de fois que d'interruptions inhibées durant la boucle FOR-NEXT.

8.3. L'instruction MOD

La puissance du micro ordinateur CPC tient entre autres à des instructions qui permettent d'exécuter une série d'instructions. Le code source n'est pas plus court pour autant mais plus lisible.
Par exemple, nous allons afficher les codes ASCII de 33 à 255, en quatre colonnes à l'écran.

```
10 for i=33 to 255
20 print i;' ';chr$(i),
30 if i/4=int(i/4) then print
40 next
```

Les lignes 10 et 40 constituent la boucle. En ligne 20, la variable courante et le caractère correspondant sont affichés. La virgule après la ligne print conduit à un affichage continu. Afin de n'afficher que les quatre colonnes, il faut exécuter un print sans virgule après quatre print suivis de virgules pour réaliser le retour-chariot. En ligne 30, chaque retour est effectué avec la variable INTeger. Il est plus simple et plus lisible de réaliser cela avec la fonction MOD dont le résultat est le reste de la division :

```
4  MOD  4  = 0
6  MOD  4  = 2
10 MOD 11  = 10
```

Etant donné que le reste de la division de 4 par 4 est 0, le résultat de la fonction MOD sera 0 et la condition pour effectuer le retour chariot est remplie. On peut améliorer la ligne 30 comme suit:

```
30 if i mod 4=0 then print
```

DEUXIEME PARTIE

EXTENSIONS D'INSTRUCTIONS ET AUTRES PROGRAMMES UTILES EN LANGAGE MACHINE

IX. AIDES A LA PROGRAMMATION

9.1. La Structure des variables

Dans ce qui va suivre, nous allons nous occuper du traitement interne des variables BASIC. A la fin de cette partie, vous trouverez les extensions DUMP et XREF. DUMP affiche toutes les variables qui commencent par une lettre donnée ainsi que leurs valeurs. XREF (X=croix, en anglais : cross, c'est à dire CROSS REFERENCE) donne les lignes d'un programme BASIC dans lesquelles se trouvent des variables données. DUMP et XREF sont des extensions très utiles qui simplifient la programmation et la recherche d'erreurs.

Le BASIC 'locomotive' dispose en ce qui concerne les variables des spécificités avantageuses :

Il existe trois types de variables standards : integer (entiers), real (réels) et string (chaines de caractères). Il est agréable de pouvoir définir les variables commençant par certaines lettres avec un type particulier. Il est bon d'utiliser cette possibilité autant que possible (voir chapitre: accélération des programmes BASIC).

Dans cette classe de micro ordinateurs, c'est une exclusivité de pouvoir disposer de noms de variables avec 40 caractères significatifs. Cela signifie que tous les caractères composant le nom d'une variable la distinguent d'une autre variable. Auparavant, cela était vrai uniquement pour les deux premiers caractères. D'une part, cela donne la possibilité de définir un nombre presque infini de variables. D'autre part, et c'est plus important, les variables peuvent avoir des noms significatifs. On peut maintenant écrire 'canaldesortie' plutôt que 'cs'.

Naturellement, il ne faut pas exagérer la longueur des noms, car le programme est ralenti. Pour sauvegarder des noms de variables de longueur différente, les créateurs du BASIC 'locomotive' ont pensé à certains trucs.

Avec cela, la vitesse de traitement est augmentée à la différence de la méthode précédente, bien que cela soit un luxe, étant donné que les noms ont une longueur définie.
Avant tout, voici quelques informations sur les bases du traitements des variables par l'ordinateur.

Pour gérer les valeurs de variables, se trouve un tableau des variables de chaque programme BASIC. Dans ce tableau, on trouve les variables avec leur nom,leur type et leur valeur. Si par exemple un nom de variable est trouvé dans une formule, lors du fonctionnement d'un programme, ce nom est recherché dans le tableau et sa valeur est lue.

Dans certains BASICs, on trouve la fonction VARPTR (VARiable PoinTeuR), qui donne l'adresse d'une variable dans le tableau des variables. Cette fonction existe aussi dans le BASIC du CPC mais elle n'est pas donnée dans le manuel. Le pointeur de variable d'une variable est donné par: @nomdevariable.

La valeur obtenue ainsi à la VARPTR) donne l'adresse du tableau de variables où se trouve la valeur de la variable. La valeur est sauvegardée différemment suivant le type de la variable.

VARIABLES INTEGER

Pour les variable INT, la valeur est tout simplement constituée de l'octet de poids fort et de celui de poids faible du nombre. L'adresse la plus faible, c'est à dire celle qu'indique le pointeur contient toujours l'octet de poids fort.

```
a%=100
print peek(@a%)+256*peek(@a%+1)
```

donne 100, la valeur correspondant. Si la variable contient des nombres négatifs, la valeur obtenue avec PEEK doit être recalculée avec INT. C'est utile car il existe des nombres INT sans préfixe. -256 est une variable préfixée, &8100 est une variable non préfixée. En décimal, la dernière valeur est 20736.

Mais print &8100 donne -256.

Pour les variables préfixées, le bit 15 du nombre, normalement utilisé pour les nombres hexadécimaux, est interprèté comme préfixe.

VARIABLES REAL

Pour ce type, les valeurs sont sauvegardées sous forme exponentielle. Le nombre est fixé de telle façon qu'il ne présente qu'une seule place avant la virgule. La véritable position de la virgule est donnée avec l'exposant:

54321=5.4321*10^4

De toute manière, cela est effectué de façon binaire et non décimale par l'ordinateur.

Cinq octets sont utilisés pour sauvegarder une variable réelle. Les quatre premiers octets constituent la mantisse, c'est à dire la partie numérique du nombre, pour laquelle la virgule est toujours située en seconde position. Le MSB (MOST SIGNIFICANT BIT: bit de plus haute valeur) du quatrième octet indique le préfixe du nombre. Le cinquième octet contient l'exposant. Pour obtenir la valeur réelle du nombre, il faut soustraire 129 du cinquième octet, c'est à dire :

l'exposant est fixé avec 129 additionné.

Comme nous l'avons indiqué plus haut, la valeur est sauvegardée en binaire. La mantisse est donc constituée de zéros et de uns. Comme la mantisse a une place avant la virgule et que zéro est un chiffre sans valeur avant la virgule, il ne reste plus que le un. C'est à dire que le binaire précédant la virgule est toujours un. Par conséquent, le un n'a pas besoin d'être précisé et à sa place, on donne le préfixe (bit 7 de l'octet 4). Comme pour les nombre INT, l'octet de poids fort prend la valeur la plus faible et les octets de plus faible adresse ont aussi les plus faibles valeurs. Les quatre octets de mantisse sont appelé m1 à m4.

On obtient la mantisse avec:

```
print (m1+256*m2+256^2*m3+256^3*(m4 or 128))/256^4
```

'or 128' donne le 1 sauvegardé. La division par 256^4 est indispensable pour obtenir un nombre avec une place avant la virgule.

La valeur d'un nombre réel peut être calculée avec le VARPTR à l'aide du petit programme suivant.

```
100 a=13:'variable à virgule flottante recherchée
110 ad=@a:'adresse de a
120 m1=peek(ad):m2=peek(ad+1):m3=peek(ad+2)
130 m4=peek(ad+3):ex=peek(ad+4)
140 print (1-2*sgn(m4 and 128))*2^(ex-1299*(1+(m4 and
127)+(m3+(m2+m1/256)/256)/128)
```

VARIABLES STRING

Le dernier groupe de variables est celui qui peut contenir des données alphanumériques. Une chaîne est en fait une série d'octets qui contiennent les codes ASCII des caractères considérés. Comme la longueur des chaines peut varier de 0 à 255, les contenus (les codes) ne sont pas sauvegardés directement dans le tableau. Pour la sauvegarde des chaines, il existe un secteur particulier en RAM (vers la fin de la RAM BASIC). Le tableau de variable contient seulement l'adresse de départ de la chaîne dans le tableau des chaines, ainsi que la longueur de la chaîne. Les deux valeurs constituent le descripteur de chaîne. Le programme qui suit explique ce processus.

```
100 input a$
110 ad=@a$
120 i=peek(ad)
130 stad=peek(ad+1)+256*peek(ad+2)
140 for i=stad to stad+i-1: print chr$(peek(i));:next i
```

Nous avons maintenant expliqué tous les types de variables. Voyons maintenant la structure du tableau de variables.

LE TABLEAU DE VARIABLES

L'adresse de départ du tableau de variables peut être lue avec PEEK dans la RAM du système. A L'adresse &ae85/6 (664,6128:&ae68/9) se trouve l'adresse de départ du tableau de variables. le tableau de variables est construit selon la structure suivante:

nombre d'octets	signification
2	adresse de chaînage (nom de variable sans
jusqu'à 39	la dernière lettre), dernière lettre+128
1	type de la variable
2,3 ou 5	valeur de variable
	début de la donnée suivante
	avec la même structure

Voici la signification de l'adresse de chaînage: le nom de la variable est sauvegardé principalement en tant que chaîne, c'est à dire sous forme ASCII. Malheureusement, cela n'est pas toujours le cas. Les chiffres inclus dans le nom de variable ne sont pas sauvegardés selon leur code ASCII.

Pour préciser la fin du nom de variable, le bit 7 de la dernière lettre est fixé, c'est à dire que l'on ajoute 128 au code. Le type de la variable est indiqué par un chiffre. On a :

1 INTEGER
2 STRING
3 REAL

Comme vous le savez, les majuscules et les minuscules ne sont pas distinguées dans un nom de variable. Les noms de variables sont toujours sauvegardés avec des majuscules. La modification des minuscules se fait par l'effacement du bit 5 du code. Essayez:

```
PRINT CHR$(ASC(b) AND NOT 2^5)
```

Avec ces opérations, les minuscules sont transformées en majuscules. Si il y a un chiffre dans le nom de la variable, la transformation ne fonctionne évidemment pas. Le résultat est un code inférieur à 32. Avant l'affichage du code du caractère, il faut donc fixer le bit 5 avec OR &20. Le nom est alors donné avec des minuscules et les chiffres corrects. OR &20 transforme les majuscules en minuscules.

Les chiffres précisant le type qui sont utilisés dans le tableau de variable sont égaux à la longueur de la valeur de la variable moins un. (2,3 ou 5 octets)

La valeur, dont la longueur fait comme indiqué 2,3 ou 5 octets contient la valeur (pour les nombres) ou l'adresse de la valeur (pour les chaines) selon le codage indiqué précédemment.
Avec cette structure, il serait possible de rechercher une variable dans le tableau, en comparant le nom de variable avec ceux sauvegardés. Mais cela prendrait trop de temps, surtout pour les longs programmes.

La recherche d'un nom de variable dans le tableau est considérablement accèlérée avec l'utilisation de listes chainées. Pour rendre cela possible, on utilise l'adresse de chaînage au début de chaque série de données. Le principe de la gestion de données avec des listes chaînées est que à chaque série de donnée, dans notre cas, la valeur des variables, est assigné un pointeur qui indique la prochaine série de donnée. C'est l'adresse de chaînage.

Pour le CPC, toutes les variables qui commencent avec la même lettre sont chaînées entre elles de cette manière. Avec cela, le temps de recherche est réduit au 1/26 5me, car il y a 26 listes chaînées indépendantes (une par lettre). Les listes chaînées utilisent une plus grande place. Cela a été calculé au mieux. Pour gérer une liste chaînée, il faut tenir compte de deux faits:

D'abord l'adresse du premier et du dernier élément de la liste est utilisée. Ces éléments ne sont reliés à aucun autre et leurs adresses doivent être mémorisées séparément. L'adresse du dernier élément n'est pas utile pour la méthode utilisée ici, car on décide que si l'adresse de chaînage prend la valeur 0, le dernier élément est atteint. L'adresse du premier élément de la liste est mémorisée dans la RAM à l'emplacement courant.

La recherche d'une variable est menée selon le schéma suivant :

1) Prendre la première lettre du nom
2) Prendre l'adresse de départ du premier élément correspondant à la lettre.
3) Lire l'adresse de chaînage suivante et la mémoriser.
4) Tester la ressemblance des noms de variables.
5) Si les noms de variables sont différents, alors revenir au 3) avec l'adresse mémorisée (si différente de 0), si l'adresse de chaînage est 0, alors la variable est introuvable.
6) Si les noms de variables sont semblables, alors la variable a été trouvée.

Comme le tableau de variable doit pouvoir être déplacé, car il est repoussé à chaque transformation du programme en BASIC, les adresses de chaînage sont toujours mémorisées comme la différence jusqu'à l'adresse de départ du tableau.

Le programme DUMP est bati sur cette structure de listes chaînées du tableau de variables.

9.2. DUMP. Affichage des valeurs de variables

DUMP donne toutes les variables qui commencent avec la lettre précisée, ainsi que leurs valeurs.

Comme DUMP est relié avec RSX (voir chapitre RSX pour plus d'informations), il faut entrer shift+@ avant le mot DUMP.
Si toutes les variables doivent être affichées, il suffit alors de taper la touche RETURN. Pour obtenir certains secteurs de lettres, il faut entrer "' (shift+7) auparavant. Le secteur de variable suit alors, comme pour les ordres DEFINT, DEFREAL et DEFSTR. Une instruction complète par exemple est :

```
|DUMP'a-c,f-l,x,y
```

Le listing assembleur du programme sert aussi pour l'instruction XREF. Ne vous laissez pas décontenancer et ignorez les lignes concernant Xref, nous en parlerons plus tard.

Pour effectuer des opérations arithmétiques à l'intérieur de l'ordinateur, il y a le FAC (accumulateur à virgule flottante). Comme le calcul avec des nombres réels utilise 5 octets, la mémorisation dans les registres n'est pas nécessaire. Cela signifie que toutes les opérations concernant les nombres peuvent être faites avec FAC. Le FAC est une zone en mémoire RAM qui est longue de 6 octets. Le premier octet du FAC contient le type de la variable courante (adresse &b0c1 pour le 464). Les octets qui suivent contiennent la valeur dans l'ordre donné. Le descripteur de type dans FAC correspond exactement au nombre d'octets nécessaires pour mémoriser la valeur, c'est à dire 2 pour INT, 3 pour STRING, 5 pour REAL. Le FAC est utilisé pour afficher la valeur avec la routine 'PRINT FAC'

Le listing assembleur est très documenté pour être parfaitement lisible au nouveaux initiés.

```
A000            10                  ; XREF
A000            20                  ; [Cross]-REFference
A000            30                  ; affiche les lignes BASIC
A000            40
; dans lesquelles se trouvent des
A000            50                  ; variables sprecifiques
A000            60                  ; Format :
A000            70                  ; "|xref'<secteurdelettres>
A000            80                  ; z.B : "|xref'c,f,i,w-z
A000            90                  ;
A000           100                  ; DUMP
A000           110
; donne toutes les variables definies
A000           120                  ; ainsi que leur valeur
A000           130                  ; Format :
A000           140                  ; "|dump'<secteurdelettres>
A000           150                  ; voir exemple plus haut
A000           160                  ;
A000           170                  ; impression : POKE &ac06,8
A000           180
; (464: poke &ac21,8) directement
A000           190                  ; avant l'instruction
A000           200
A000           210                  ; liaison avec RSX
A000 010000    220   DEFRSX LD   bc,tabrsx
A003 210000    230          LD   hl,kernal
A006 CDD1BC    240          CALL &bcd1 ; Log en extension
A009 3EC9      250          LD   a,&c9 ; RET
A00B 3200A0    260          LD   (defrsx),a
; eviter la redefinition
A00E C9        270          RET
**** Ligne  220  : TABRSX=&A00F
A00F 0000      280   TABRSX DW   table
A011 C30000    290          JP   xref
A014 C30000    300          JP   dump
**** Ligne  280  : TABLE=&A017
A017 585245    310   TABLE  DM   "XRE"
A01A C6        320          DB   &c6 ; "F"+&80
```

```
A01B 44554D   330          DM   "DUM"
A01E D0       340          DB   &d0 ; "P"+&80
A01F 00       350          DB   0
**** Ligne  230  : KERNAL=&A020
A020          360   KERNAL DS   4
A024          370
**** Ligne  290  : XREF=&A024
A024 3E01     380   XREF   LD   a,1
A026 320000   390   AUFRUF LD   (prgken),a
A029 DF       400          RST  &18 ; appel avec  Far Call,
A02A 0000     410          DW   vektor
; car la ROM susjacente doit etre
A02C C9       420          RET   ; activee
A02D          430
**** Ligne  410  : VEKTOR=&A02D
A02D 0000     440   VEKTOR DW   start
A02F FD       450          DB   253 ; LROM off / UROM on
A030          460
**** Ligne  300  : DUMP=&A030
A030 3E00     470   DUMP   LD   a,0
A032 18F2     480          JR   aufruf
A034          490
A034          500
;  CPC      6128    464    664
A034          510   ALBAPC EQU  &ae58
; &ae75 , &ae58  variable temporaire pour le PC BASIC
A034          520   BASPC  EQU  &ae1d
; &ae36 , &ae1d  Original BASIC PC
A034          530   TESBUC EQU  &ff92
; &ff71 , &ff92  Test des lettres et UPPER
A034          540   SYNTER EQU  &d0d7
; &d07b , &d0da  affichage de Syntax error
A034          550   GTVPTR EQU  &d619
; &d5db , &d61c  prend le pointeur du tableau de variables
A034          560   BADOBC EQU  &e9b9
; &e8ff , &e9be  parcourt le programme BASIC et se branche s
i necessaire a l'adresse BC
A034          570   CHKBRK EQU  &c472
; &c43c , &c475  check for Break
A034          580   LNFEED EQU  &c398
```

```
; &c34e , &c39b  sortie d'un Line Feed
A034            590  SKPCMD EQU  &e9fd
; &e943 , &ea02  skip command
A034            600  VAINIT EQU  &d6ec
; &d6b3 , &d6ef  met la variable dans la table de variables
A034            610  CHRGET EQU  &de2c
; &dd3f , &de31  charge l'octet suivant
A034            620  CHRGOT EQU  &de37
; &dd4a , &de3c  lit l'octet precedent
A034            630  CHRNEX EQU  &de25
; &dd37 , &de2a  teste l'octet suivant
A034            640  CHKKOM EQU  &de41
; &dd55 , &de46  test si virgule
A034            650  PRINT  EQU  &c3a0 ; &c356 , &c3a3
A034            660  PTLNNU EQU  &ef44
; &ee79 , &ef49  print line number
A034            670  VARFAC EQU  &ff6c
; &ff4b , &ff6c  copie la variable dans FAC
A034            680  PRTFAC EQU  &f2d5
; &f236 , &f2da  print FAC
**** Ligne  390  : PRGKEN=&A034
A034            690  PRGKEN DS   2
A036            700  WRTADR DS   2
A038            710
**** Ligne  440  : START=&A038
A038 3A34A0     720  START  LD   a,(prgken)
A03B FE00       730         CP   0 ; Dump ?
A03D 28FE       740         JR   z,weiter
A03F 010000     750         LD   bc,initva
; recherche de toutes les variables
A042 CDB9E9     760         CALL badobc ; du programme BASIC
**** Ligne  740  : WEITER=&A045
A045 2A58AE     770  WEITER LD   hl,(albapc)
A048 0641       780         LD   b,&41 ; "A" Default Start
A04A 0E5A       790         LD   c,&5a ; "Z" Default End
A04C CD37DE     800         CALL chrgot ; fin ???
A04F B7         810         OR   a ; si oui
A050 28FE       820         JR   z,anfang
; alors commencer avec la valeur par defaut
A052 23         830         INC  hl ; PC sur "'"
```

```
A053 CD25DE   840          CALL chrnex
; Test du caractere suivant
A056 C0       850          DB   &c0
; le caractere doit etre  "'"
A057 7E       860   VONVOR LD   a,(hl) ; lecture de la lettre
A058 CD92FF   870          CALL tesbuc
A05B 38FE     880          JR   c,ok ; la lettre est ok
A05D C3D7D0   890   ERROR  JP   synter
**** Ligne  880  : OK=&A060
A060 47       900   OK     LD   b,a
; la lettre est la valeur de depart
A061 4F       910          LD   c,a
; et aussi la valeur par defaut
A062 CD2CDE   920          CALL chrget
A065 FE2D     930          CP   &2d ; "-" ??
A067 20FE     940          JR   nz,fangan
A069 CD2CDE   950          CALL chrget
A06C CD92FF   960          CALL tesbuc
A06F 30EC     970          JR   nc,error ; si aucune lettre
A071 4F       980          LD   c,a
; la lettre est la derniere
A072 CD2CDE   990          CALL chrget
**** Ligne  940  : FANGAN=&A075
A075 E5       1000  FANGAN PUSH hl ; sauvegarde du PC
A076 CD0000   1010         CALL anfang
A079 E1       1020         POP  hl ; PC
A07A CD41DE   1030         CALL chkkom ; test si virgule
A07D 38D8     1040         JR   c,vonvor
A07F 2258AE   1050         LD   (albapc),hl
A082 C9       1060         RET
A083          1070
**** Ligne  820  : ANFANG=&A083
**** Ligne  1010  : ANFANG=&A083
A083 79       1080  ANFANG LD   a,c ; fin
A084 90       1090         SUB  b ; moins debut
A085 38D6     1100         JR   c,error
; fin<debut alors Syntax Error
A087 78       1110  NEXBUC LD   a,b
; faire XREF de la variable avec la lettre suivante
A088 04       1120         INC  b ; pour la fois suivante
```

```
A089 C5       1130          PUSH bc ; augmenter et sauvegarder
A08A CD19D6   1140          CALL gtvptr
; prend l'adresse de pointeur de la lettre
A08D 7E       1150   GLEBUC LD   a,(hl) ; HL prend la valeur
A08E 23       1160          INC  hl
; de la difference entre le debut
A08F 66       1170          LD   h,(hl)
; de la table des variables
A090 6F       1180          LD   l,a ; et la variable
A091 B4       1190          OR   h ; si la difference est 0
A092 20FE     1200          JR   nz,namaus
; aucune variable n'a la
A094 C1       1210          POP  bc ; lettre de debut
A095 79       1220          LD   a,c ; position courante (b)
A096 B8       1230          CP   b
; comparaison avec la fin (c)
A097 30EE     1240          JR   nc,nexbuc
; si pas encore fin, alors essayer la lettre suivante
A099 C9       1250          RET    ; sinon fin
A09A          1260
**** Ligne  1200  : NAMAUS=&A09A
A09A 09       1270   NAMAUS ADD  hl,bc
; Var.tab.start+Difference
A09B E5       1280          PUSH hl
; est l'adresse de l'adresse de chainage
A09C C5       1290          PUSH bc
; sauvegarder Var.tab.start
A09D 23       1300          INC  hl ; chainage
A09E 23       1310          INC  hl ; saut
A09F 7E       1320   AUSGA  LD   a,(hl)
; lecture de la lettre
A0A0 23       1330          INC  hl
A0A1 F5       1340          PUSH af
A0A2 E67F     1350          AND  &7f
; effacer le bit 7 pour l'affichage
A0A4 FE20     1360          CP   32
; un chiffre dans le nom de variable ?
A0A6 30FE     1370          JR   nc,allkla
; non, alors tout est ok
A0A8 F620     1380          OR   &20
```

```
; positionner le bit 5 du chiffre
**** Ligne  1370  : ALLKLA=&A0AA
A0AA CDA0C3  1390  ALLKLA CALL print
A0AD F1      1400         POP  af ; valeur lue
A0AE 17      1410         RLA   ; Test si bit 7 positionne
A0AF 30EE    1420         JR   nc,ausga
; non, alors continuer l'affichage
A0B1 3E20    1430         LD   a,&20 ; caractere vide
A0B3 CDA0C3  1440         CALL print
A0B6 7E      1450         LD   a,(hl)
; caractere de controle du type
A0B7 23      1460         INC  hl
A0B8 C601    1470         ADD  a,1
; nombre d'octets=type pour FAC
A0BA CD0000  1480         CALL typtes
A0BD F5      1490         PUSH af
A0BE 3A34A0  1500         LD   a,(prgken)
A0C1 B7      1510         OR   a
A0C2 CA0000  1520         JP   z,fordum ; continuer si DUMP
A0C5 F1      1530         POP  af
A0C6 C1      1540         POP  bc
A0C7 B7      1550         OR   a
A0C8 ED42    1560         SBC  hl,bc
; Adresse de la variable
A0CA 2236A0  1570         LD   (wrtadr),hl
; sauvegarder la valeur
A0CD C5      1580         PUSH bc
; pour une comparaison suivante
A0CE 010000  1590         LD   bc,suchva
; routine de recherche de variable
A0D1 CDB9E9  1600         CALL badobc
; parcourt le programme BASIC
A0D4 CD72C4  1610  SOFERT CALL chkbrk ; Break Test
A0D7 CD98C3  1620         CALL lnfeed ; Line Feed
A0DA C1      1630         POP  bc ; deb/fin lettre
A0DB E1      1640         POP  hl ; adresse de chainage
A0DC 18AF    1650         JR   glebuc
; variable suivante avec la meme lettre de debut
A0DE         1660
A0DE         1670
```

```
; routine d'initialisation des variables
**** Ligne  750  : INITVA=&A0DE
A0DE E5       1680  INITVA PUSH hl ; PC
A0DF CDFDE9   1690         CALL skpcmd
; saute une instruction
A0E2 D1       1700         POP  de
A0E3 FE02     1710         CP   2 ; fin ?
A0E5 D8       1720         RET  c ; oui, ligne suivante
A0E6 FE0E     1730         CP   &e ; Variable ??
A0E8 30F4     1740         JR   nc,initva
; non, alors continuer la recherche
A0EA FE07     1750         CP   7
A0EC 28F0     1760         JR   z,initva
A0EE FE08     1770         CP   8
A0F0 28EC     1780         JR   z,initva
A0F2 EB       1790         EX   de,hl
A0F3 D5       1800         PUSH de ; PC nach skip
A0F4 CD2CDE   1810         CALL chrget
A0F7 CDECD6   1820         CALL vainit
; recherche du nom et chargement de la vriable
A0FA E1       1830         POP  hl
A0FB 18E1     1840         JR   initva
A0FD          1850
A0FD          1860
; routine de recherche de variable
**** Ligne  1590  : SUCHVA=&A0FD
A0FD E5       1870  SUCHVA PUSH hl ; PC
A0FE CDFDE9   1880         CALL skpcmd
; saute une instruction
A101 D1       1890         POP  de
A102 FE02     1900         CP   2 ; fin ?
A104 D8       1910         RET  c ; oui, alors ligne suivante
A105 FE0E     1920         CP   &e ; Variable ??
A107 30F4     1930         JR   nc,suchva
; non, alors on continue la recherche
A109 FE07     1940         CP   7
A10B 28F0     1950         JR   z,suchva ; non
A10D FE08     1960         CP   8
A10F 28EC     1970         JR   z,suchva
A111 EB       1980         EX   de,hl
```

```
A112 D5       1990          PUSH de
A113 CD2CDE   2000          CALL chrget
A116 23       2010          INC  hl
; HL sur l'adresse de debut de la variable
A117 5E       2020          LD   e,(hl)
A118 23       2030          INC  hl
A119 56       2040          LD   d,(hl)
A11A 2A36A0   2050          LD   hl,(wrtadr)
; comparaison avec
A11D B7       2060          OR   a
A11E ED52     2070          SBC  hl,de ; l'adresse courante
A120 E1       2080          POP  hl ; PC apres SKIP
A121 20DA     2090          JR   nz,suchva
; different, alors continuer la recherche
A123 E5       2100          PUSH hl ; Adr.Typ Prog.
A124 2A1DAE   2110          LD   hl,(baspc) ; lire BASIC PC
A127 7E       2120          LD   a,(hl)
A128 23       2130          INC  hl
A129 66       2140          LD   h,(hl)
A12A 6F       2150          LD   l,a
A12B CD44EF   2160          CALL ptlnnu ; Print Line Number HL
A12E 3E20     2170          LD   a,&20
A130 CDA0C3   2180          CALL print
A133 E1       2190          POP  hl
A134 18C7     2200          JR   suchva
A136          2210
**** Ligne  1520  : FORDUM=&A136
A136 F1       2220   FORDUM POP  af ; suite de DUMp
A137 FE03     2230          CP   3
A139 20FE     2240          JR   nz,nostri
A13B 46       2250          LD   b,(hl)
; longueur de la chaine
A13C 97       2260          SUB  a ; effacer l'accu
A13D B8       2270          CP   b ; longueur=0 ?
A13E 28FE     2280          JR   z,skip
; alors ne pas afficher
A140 E5       2290          PUSH hl ; adresse de descripteur
A141 23       2300          INC  hl
A142 7E       2310          LD   a,(hl) ; low Byte
A143 23       2320          INC  hl
```

```
A144 66      2330         LD   h,(hl) ; high Byte
A145 6F      2340         LD   l,a
A146 7E      2350  NESTCH LD   a,(hl)
A147 23      2360         INC  hl
A148 CDA0C3  2370         CALL print
A14B 10F9    2380         DJNZ nestch
A14D E1      2390         POP  hl
**** Ligne  2280  : SKIP=&A14E
A14E 3E03    2400  SKIP   LD   a,3 ; Longueur du descripteur
A150 18FE    2410         JR   skval
**** Ligne  2240  : NOSTRI=&A152
A152 E5      2420  NOSTRI PUSH hl
A153 CD6CFF  2430         CALL VARFAC
; fixer le type et VAR (HL) -> FAC
A156 E1      2440         POP  hl
A157 CDD5F2  2450         CALL prtfac ; print FAC
**** Ligne  2410  : SKVAL=&A15A
A15A C3D4A0  2460  SKVAL  JP   sofert
A15D         2470
**** Ligne  1480  : TYPTES=&A15D
A15D F5      2480  TYPTES PUSH af ; donne le type
A15E E5      2490         PUSH hl
; caractere correspondant a sortir
A15F E607    2500         AND  7
; ne prendre en compte que les bit 0-2
A161 EE27    2510         XOR  &27
A163 FE22    2520         CP   &22
A165 20FE    2530         JR   nz,ok1
A167 D601    2540         SUB  1
**** Ligne  2530  : OK1=&A169
A169 CDA0C3 `2550  OK1    CALL print
A16C 3E20    2560         LD   a,&20 ; " "
A16E CDA0C3  2570         CALL print
A171 E1      2580         POP  hl
A172 F1      2590         POP  af
A173 C9      2600         RET

Programme :xdump
Debut : &A000    Fin : &A173
Longueur : 0174
```

0 Erreur

Table de variables :

DEFRSX	A000	TABRSX	A00F	TABLE	A017	KERNAL	A020
XREF	A024	AUFRUF	A026	VEKTOR	A02D	DUMP	A030
ALBAPC	AE58	BASPC	AE1D	TESBUC	FF92	SYNTER	D0D7
GTVPTR	D619	BADOBC	E9B9	CHKBRK	C472	LNFEED	C398
SKPCMD	E9FD	VAINIT	D6EC	CHRGET	DE2C	CHRGOT	DE37
CHRNEX	DE25	CHKKOM	DE41	PRINT	C3A0	PTLNNU	EF44
VARFAC	FF6C	PRTFAC	F2D5	PRGKEN	A034	WRTADR	A036
START	A038	WEITER	A045	VONVOR	A057	ERROR	A05D
OK	A060	FANGAN	A075	ANFANG	A083	NEXBUC	A087
GLEBUC	A08D	NAMAUS	A09A	AUSGA	A09F	ALLKLA	A0AA
SOFERT	A0D4	INITVA	A0DE	SUCHVA	A0FD	FORDUM	A136
NESTCH	A146	SKIP	A14E	NOSTRI	A152	SKVAL	A15A
TYPTES	A15D	OK1	A169				

Il y a une particularité de la table des variables dont nous n'avons pas encore parlé.

Les fonctions qui sont définies avec DEF FN sont également sauvegardées dans la table des variables. La marque de type d'une définition de fonction est &41, &42 ou &44. Cela signifie que le bit 6 veut dire "fonction", et l'octet faible (1,2 ou 4) indique de la manière que nous savons, de quel type est le résultat de la fonction.

Tous les noms de fonction sont liés entre eux à travers des adresses de chaînage. A l'adresse &AE04/5 (664/6128: &ADEB/C) figure l'adresse du premier élément de la liste. La "valeur" d'un nom de fonction se compose de deux octets qui indiquent dans quel endroit du programme BASIC figure la définition de fonction.

Les adresses RAM du premier élément pour chaque lettre initiale figurentauxadresses&ADD0à&AE03(pour664/6128:&ADB7à&ADEA), deux octets étant utilisés pour chaque lettre. Lorsque vous utilisez ces pointeurs, veillez à ne pas indiquer l'adresse elle-même mais la différence entre l'adresse et l'adresse de départ de la table des variables, &AE85/6 (664/6128: &AE68/9).

9.3. XREF (Cross REFerence)

La syntaxe de l'instruction XREF est identique à celle de l'instruction DUMP. Comme de grandes sections de programme sont utilisées aussi bien par DUMP que par XREF, ces deux instructions ont été réalisées dans le même programme. Comme pour DUMP, les boucles NEXBUC et GLEBUC cherchent les variables nécessaires dans la table.

Tout au début du programme, XREF utilise une routine système très intéressante, pour entrer toutes les variables dans la table des variables. A l'adresse &E8FF (6218: &E9B9/ 664: &E9BE) figure la routine BADOBC qui parcourt un programme BASIC ligne par ligne. Sa particularité est que, dès que le début d'une ligne ait été trouvée elle peut appeler n'importe quelle autre routine qui se chargera alors d'examiner la ligne trouvée.

L'adresse de la routine qui doit faire la recherche est sauvée dans le registre BC lors de l'appel de BADOBC. Dans le programme XREF, c'est d'abord la routine INITVA puis la routine SUCHVA.

Pour comprendre la routine SUCHVA, il faut voir comment est construit un programme BASIC et en particulier l'aspect des variables.

Avant chaque variable se trouve un chiffre de reconnaissance. Le chiffre donne le type de la variable et indique si le descripteur de type a été donné (%,$ ou !). pour cela :

02 integer avec %
03 string avec $
04 real avec !
0b integer
0c string
0d real

Après le chiffre de reconnaissance il y a la différence entre l'adresse de la variable dans le tableau et l'adresse de départ du tableau. Puis après cet indicateur d'adresse, il y a le nom de variable où le bit 7 de la dernière lettre est fixé.

SUCHVA teste d'abord s'il s'agit d'un chiffre de reconnaissance, si oui le nom est comparé puis le chiffre de reconnaissance est transformé puis comparé. Si cela correspond, le numéro de la ligne BASIC courante est affiché.

Pour le programmeur qui ne possède pas l'assembleur du livre 'le langage machine du CPC' ou un autre, voici un chargeur BASIC du programme.

La liaison des nouvelles instructions avec RSX est effectuée avec CALL &A000. Ensuite, les instructions DUMP et XREF peuvent être utilisées selon le format décrit.

```
10 ' XREF et DUMP pour 464
20 ' liaison RSX avec call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 49982 THEN PRINT"Erreur dans les Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA FC,A6,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,5F,03
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,FF,E8,2A,75,AE
170 DATA 06,41,0E,5A,CD,4A,DD,B7
180 DATA 28,31,23,CD,37,DD,C0,7E
190 DATA CD,71,FF,38,03,C3,7B,D0
200 DATA 47,4F,CD,3F,DD,FE,2D,20
210 DATA 0C,CD,3F,DD,CD,71,FF,30
220 DATA EC,4F,CD,3F,DD,E5,CD,83
230 DATA A0,E1,CD,55,DD,38,D8,22
240 DATA 75,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,DB,D5,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,56,C3,F1,17,30
300 DATA EE,3E,20,CD,56,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,FF,E8,CD,3C,C4,CD
350 DATA 4E,C3,C1,E1,18,AF,E5,CD
360 DATA 43,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,3F,DD,CD
390 DATA B3,D6,E1,18,E1,E5,CD,43
```

```
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,3F,DD,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,36,AE,7E
450 DATA 23,66,6F,CD,79,EE,3E,20
460 DATA CD,56,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,56,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,4B,FF,E1,CD
510 DATA 36,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,56,C3,3E,20,CD,56
540 DATA C3,E1,F1,C9
```

```
10 ' XREF et DUMP pour 664
20 ' liaison RSX avec call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50380 THEN PRINT"Erreur dans les Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 38,A0,3E,0D,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,E1,D0,18,D8
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,BE,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,3C,DE,B7
180 DATA 28,31,23,CD,2A,DE,C0,7E
```

```
190 DATA CD,92,FF,38,03,C3,DA,D0
200 DATA 47,4F,CD,31,DE,FE,2D,20
210 DATA 0C,CD,31,DE,CD,92,FF,30
220 DATA EC,4F,CD,31,DE,E5,CD,83
230 DATA A0,E1,CD,46,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,1C,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A3,C3,F1,17,30
300 DATA EE,3E,20,CD,A3,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,BE,E9,CD,75,C4,CD
350 DATA 9B,C3,C1,E1,18,AF,E5,CD
360 DATA 02,EA,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,31,DE,CD
390 DATA EF,D6,E1,18,E1,E5,CD,02
400 DATA EA,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,31,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,49,EF,3E,20
460 DATA CD,A3,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A3,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA DA,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A3,C3,3E,20,CD,A3
540 DATA C3,E1,F1,C9
```

```
10 ' XREF et DUMP pour 6128
20 ' liaison RSX avec call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50608 THEN PRINT"Erreur dans les Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 20,A0,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,D1,00
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,B9,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,37,DE,B7
180 DATA 28,31,23,CD,25,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,D7,D0
200 DATA 47,4F,CD,2C,DE,FE,2D,20
210 DATA 0C,CD,2C,DE,CD,92,FF,30
220 DATA EC,4F,CD,2C,DE,E5,CD,83
230 DATA A0,E1,CD,41,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,19,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A0,C3,F1,17,30
300 DATA EE,3E,20,CD,A0,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,B9,E9,CD,72,C4,CD
350 DATA 98,C3,C1,E1,18,AF,E5,CD
360 DATA FD,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,2C,DE,CD
390 DATA EC,D6,E1,18,E1,E5,CD,FD
```

```
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,2C,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,44,EF,3E,20
460 DATA CD,A0,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A0,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA D5,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A0,C3,3E,20,CD,A0
540 DATA C3,E1,F1,C9
```

X. CREATION DE LIGNES BASIC A PARTIR DU BASIC

Une instruction qui manquait jusqu'à maintenant à tous les BASIC était la possibilité de créer des lignes BASIC à partir d'un programme.

En cela, cette instruction offre des aspects totalement nouveaux dans la programmation en BASIC.

Existe-t-il une instruction qui permette de créer des lignes de programme à partir d'un programme ?

Si c'est possible, on peut imaginer des programmes qui créent des programmes qui créent des programmes... A quoi bon apprendre à programmer si les ordinateurs peuvent le faire ?

Heureusement, on n'est pas encore allé aussi loin. Cependant, l'instruction de création de programmes offre des possibilités intéressantes. Nous allons développer cette instruction.

Lors de la saisie de lignes BASIC en mode direct, celles ci sont mémorisées dans un tampon presque comme des chaînes de caractères jusqu'à ce qu'intervienne un return. Ensuite, la chaîne est interprétée en tant que ligne BASIC (reconnaissable à son numéro de ligne!). Si cela fonctionne, la chaîne est transformée dans le format interne des lignes BASIC. Durant cette opération, les instructions sont transformées en token et mémorisées sous cette forme. Enfin, la ligne BASIC est introduite dans le programme existant en fonction de son numéro de ligne. Les lignes dont le numéro est plus grand sont repoussées vers le haut.

Pour mettre au point notre idée, nous utilisons les mêmes routines que celles décrites précédemment. Nous avons besoin de l'adresse de la chaîne qui contient la ligne à introduire, qui doit être donnée par l'utilisateur avec @chainedecaractère. Lors de la saisie d'une seule valeur, celle ci se trouve dans le registre DE, après un appel par CALL ou des extensions RSX. DE contient donc l'adresse du descripteur de chaîne de la chaîne à transformer.

Après la lecture de la longueur de la chaîne dans le descripteur de chaîne, l'adresse de la chaîne est lue et chargée dans le registre HL.

Avec la routine CHRSKP, les caractères vides et les caractères de déplacement du curseur sont lus éventuellement. Si un octet vide est rencontré, TESTER teste si il y a un numéro de ligne au début de la ligne. Si oui, la ligne est traduite avec ASSEMB puis ajoutée au programme. La fin de la ligne est reconnue avec les octets nuls.

Voici maintenant le listing assembleur et le chargeur BASIC correspondant au programme que nous venons de décrire.

```
A000            10                  ; liner
A000            20                  ; cree des lignes de BASIC
A000            30                  ; 1. en mode direct
A000            40                  ; 2. dans un programme si
A000            50
; n0 de ligne > a la ligne courante
A000            60                  ; a$ = ligne en ASCII-
A000            70                  ; Code, Fin = octet nul
A000            80                  ; Format : call &a000,@a$
A000            90
A000           100           ORG   &a000
A000           110
;   CPC     6128 ;   464  ,  664
A000           120  CHRSKP EQU   &de4d ; &dd61  , &de52
A000           130  TESTER EQU   &eecf ; &ee04  , &eed4
A000           140  ASSEMB EQU   &e7a5 ; &e6c6  , &e7aa
A000 DF        150           RST   &18 ; Far Call
A001 0000      160           DW    vektor
; etant donne que la ROM BASIC
A003 C9        170           RET    ; doit etre activee
**** Ligne  160  : VEKTOR=&A004
A004 0000      180  VEKTOR DW    start
A006 FD        190           DB    253 ; low ROM off, upp ROM on
**** Ligne  180  : START=&A007
A007 EB        200  START  EX    de,hl
; adresse donnee apres HL
A008 23        210           INC   hl
; lecture de la longueur de la chaine
A009 5E        220           LD    e,(hl)
; adresse de la chaine en low byte
A00A 23        230           INC   hl
A00B 56        240           LD    d,(hl)
; adresse de la chaine en high byte
A00C EB        250           EX    de,hl
; adresse de la chaine apres HL
A00D CD4DDE    260           CALL chrskp
A010 B7        270           OR    a
A011 C8        280           RET   z
```

```
A012 CDCFEE   290          CALL tester
A015 D0       300          RET  nc
A016 CDA5E7   310          CALL assemb
; traduction et mise en place de la ligne; HL doit pointer s
ur le premier octet de la ligne en ASCII
A019 C9       320          RET   ; fini!!!
```

```
Programme :liner
Debut : &A000    Fin : &A019
Longueur : 001A
 0  Erreur
Table de variables :
CHRSKP DE4D  TESTER EECF  ASSEMB E7A5  VEKTOR A004
START  A007
```

```
10 REM chargeur BASIC pour liner
20 FOR i=&A000 TO &A019
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 4275 THEN PRINT"Erreur dans les Datas":END
60 '464: IF s<> 4123 THEN .....
70 '664: IF s<> 4290 THEN .....
80 PRINT"ok!":END
90 DATA DF,04,A0,C9,07,A0,FD,EB
100 DATA 23,5E,23,56,EB,CD,4D,DE
110 '664:..,..,..,..,..,..,52,DE
120 DATA B7,C8,CD,CF,EE,D0,CD,A5
130 '464:..,..,..,04,EE,..,..,c6
140 '664:..,..,..,D4,EE,..,..,AA
150 DATA E7,C9
160 '464:E6,C9
170 '664:E7,C9
```

Pour expliquer l'utilisation de l'instruction, voici un petit programme:

```
10 memory &9fff: rem liner est-il chargé?
20 numlin=100
30 z$=str$(numlin)+'rem c'est la ligne'+str(numlin)
40 z$=z$+chr$(0):rem caractère de fin
50 call &a000,@z$
60 numlin=numlin+10
70 if numlin<210 then 30
80 z$=str$(numlin)+'list'+chr$(0)
90 call &a000,@z$
```

L'utilisation du programme subit les limitations suivantes:

- le numéro de ligne de la ligne crée doit être plus grand que le numéro de la ligne où est exécutée l'instruction CALL.
- L'instruction ne doit pas se trouver dans une boucle FOR-NEXT ou WHILE-WEND.

Ces limites sont inévitables car autrement la création du programme après le CALL ne serait pas faite à la bonne place ou bien la variable ne serait pas retrouvée. La raison de cela est le déplacement du programme BASIC pour la création de la nouvelle ligne.

L'utilisation la plus simple de l'instruction est évidemment la création de ligne de data pour une gestion de données par exemple. Avec le créateur de lignes, les données peuvent être introduites directement dans le programme. Dans le livre 'le langage machine du CPC', vous trouverez le programme d'une instruction qui crée automatiquement un chargeur BASIC à partir du code résident en mémoire. Une utilisation professionnelle de l'instruction pourrait être la suivante: Une firme de logiciels vend des programmes de gestion. Pour ne pas créer un nouveau programme par client en fonction de ses besoins spécifiques, on écrit un programme de développement qui analyse toutes les possibilités. Après avoir donné les désirs, le système de développement écrit un code correspondant au cas considéré.

XI. COPIE HAUTE RESOLUTION GRAPHIQUE

Afin de pouvoir imprimer les dessins obtenus avec le programme de tracé de fonctions en trois dimensions, voici un programme de 'hardcopy'. Ce programme fonctionne sans adaptation sur l'imprimante EPSON FX-80 et compatibles. Pour l'adapter aux autres types d'imprimantes, il faut changer seulement la séquence de direction qui passe l'imprimante en mode graphique, à condition qu'il existe un mode d'impression 8 bits.

Pour la copie haute résolution, la RAM vidéo du CPC, de l'adresse &c000 à &ffff est lue en mode 2. La mémoire d'écran est batie de façon peu ordinaire sur le CPC. Mais comme sur les autres appareils, un octet correspond à l'écran à un nombre donné de points (en mode 2, 8 points). Le problème réside seulement dans le fait que l'imprimante affiche les points les uns au dessous des autres et non à coté comme sur l'écran.

Voyons comment s'affiche un caractère spécifié sur l'imprimante.

								code écran
0	0	0	0	0	0	0	0=	00
0	0	1	0	1	0	0	0=	40
0	1	0	0	0	1	0	0=	68
1	0	1	1	1	0	1	0=	186
0	1	0	0	0	1	0	0=	68
0	0	1	0	1	0	0	0=	40
0	0	0	1	0	0	0	0=	16
1	1	1	1	1	1	1	0=	254

codes imprim. 17 41 85 19 85 41 17 0

Le caractère est affiché à l'écran avec :

```
mode 2
for i=&c000 to &f800 step &800
poke i,a:next
data 16,40,68,16,68,254
```

Pour l'imprimer, il faut donner les valeurs des colonnes :

```
print#8,chr$(27);'*'chr$(1);chr$(8);chr$(0);
for i=0 to 7:read a
print#8,chr$(a);:next
data 17,41,85,19,85,41,17,0
```

La première instruction print#8 est le code d'impression pour imprimantes epson, qui permet l'impression en mode bitcopy.

Le programme de HARDCOPY doit donc adapter le format de l'écran pas à pas pour envoyer les codes des colonnes. Comme cela peut durer quelques heures en BASIC, voici une solution en langage machine. Le programme de HARCOPY utilise quelques routines système très pratiques. Nous allons d'abord les décrire rapidement.

Pour le CPC, L'impression de l'écran graphique sans adaptation de hardware est quelque chose de difficile. La raison en est que les constructeurs du CPC n'ont prévu aucune interface centronic pour l'imprimante, mais ils ont fait une sortie 7 bits. Normalement, les sorties se font en 8 bits, car un octet fait 8 bits et l'imprimante imprime des octets.

Tant que l'on ne s'occupe que d'imprimer du texte, ce n'est pas génant. Mais pour l'impression graphique, le bit manquant crée des problèmes car les dessins sont faits avec des octets.

Le programme doit donc combler ce manque, car seulement 7 points seront affichés. Mais comme la mémoire d'écran du CPC est organisée verticalement en suites de 8 octets et comme chaque caractère fait 8 bits de haut, il faut faire des calculs complexes pour imprimer seulement 7 bits à une vitesse convenable.

L'écran comprend 25 lignes de 8 rangées de points, soit 200 lignes de points. Si on travaille avec 7 lignes de points à chaque fois, il faudra imprimer 28*7 lignes de points, et il restera un problème: à la fin demeureront 4 (200-28*7) lignes. Celles ci devront être traitées spécialement. De plus, l'écran affiche 640=&280 points en horizontale.

Pour les Epson et compatibles, la mise en route du mode graphique doit s'accompagner de l'indication du nombre d'octets à recevoir. L'octet de poids faible de &280 est &80. Dans cet octet, le bit 7 est fixé, donc cette valeur ne peut pas être envoyée. Pour résoudre ce problème, l'imprimante ne recevra que &27f octets. Cela signifie que la dernière ligne de l'écran ne sera pas imprimée. C'est un défaut, mais on pourra se passer presque dans tous les cas de la dernière ligne.

Venons en au programme.

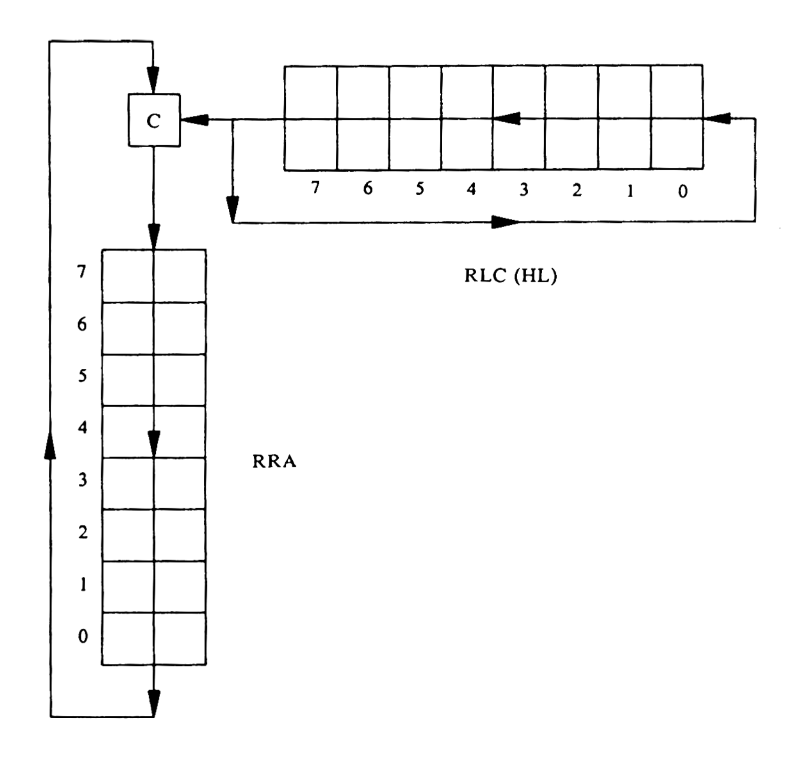
La lecture des 7 octets se fait dans un tableau dans lequel sont sauvegardées les adresses des octets courants. La huitième adresse du tableau contient toujours l'adresse du premier des sept octets suivants. Ce tableau est créé à partir du label NELI1 pour les lignes suivantes. On utilise pour cela la routine système SCR NEXT LINE. Elle augmente l'adresse d'écran dans HL, de telle façon que cette adresse soit celle de la ligne suivante. Dans la boucle NEBIT, se trouve la transformation de 8 bit en format 7 bit.

Pour cela, on charge l'adresse de l'octet voulu dans le registre HL à partir du dernier élément du tableau (TABLET). Avec RLC (HL), l'octet considéré est tourné vers la gauche et le MSB (bit significatif) est chargé dans le CARRY.

Le point du dessin est donc dans le CARRY et sera ajouté à l'accumulateur par RRA. Puis la boucle reprend du début. Après sept boucles, l'accumulateur contient les sept bits hauts de la ligne de l'écran.

Cette boucle est un bon exemple des possibilités du langage machine utilisé élégamment.

Revoyons le fonctionnement de la boucle avec un schéma:



A la fin de la boucle NEBIT, le contenu de l'accumulateur subit une nouvelle rotation. Ainsi, notre matrice de BITs ne contient plus le bit no 7, qui ne peut pas être envoyé. Enfin, l'octet créé est envoyé vers l'imprimante.

Ce processus fonctionne huit fois par octet. Ainsi, chaque bit est chargé dans l'accumulateur par rotation. Un effet parrallèle intéressant de cette méthode de 'programmation' est qu'elle montre la rotation des octets à l'écran. Souvent, on voit cela au début d'une ligne seulement car le programme doit attendre que l'imprimante soit prête.

Si tous les bits d'un octet sont envoyés, le tableau reçoit l'octet suivant avec la boucle NEBY1 qui utilise la routine système SCR NEXT BYTE. Lors du chargement de l'adresse d'écran courante dans le registre HL, SCR NEXT LINE augmente la valeur de HL, afin qu'il pointe sur l'octet suivant de la même ligne. En fonction du compteur d'octet C, l'adaptation de l'octet courant et son affichage est effectué, à condition que la fin de la ligne ne soit pas atteinte.

Si la fin de la ligne a été atteinte, le compteur E indique si il faut imprimer d'autres lignes. Si c'est le cas, on se branche sur le label NELINE. A partir de là le processus reprend du début. Si il s'agit de la dernière ligne à imprimer, (LD A,E; CP1), alors le nombre des bits à imprimer est réduit à é et le tableau ne contient que quatre adresses de début, à partir du milieu.

Ici, nous avons utilisé un truc de programmation. Après avoir défini les valeurs du tableau avec LD DE,TABMIT et LD B,4, il faut naturellement sauter l'instruction LD DE,TABANF. Normalement, cela est effectué avec l'instruction JR. Mais il est plus court d'inscrire un octet avec la valeur &21 à la place de l'instruction JR. L'ordinateur interprète l'octet &21 comme le code-op de l'instruction LD HL,nn. Ainsi, les deux octets qui suivent appartiennent à cette instruction. Le troisième octet est l'octet haut de TABANF, c'est à dire &a0. &a0 signifie AND B. Les deux instructions ainsi crées ne gènent pas le déroulement du programme et ne transforment pas le contenu de registres importants. Après ces deux instructions, la suite normale du programme est donnée par LD HL,(TABEND). L'effet désiré, c'est à dire le saut de l'instruction LD DE,TABEND est obtenu de cette manière.

Deux sous-programmes sont également très importants : TABOUT et PRINT.

TABOUT signifie affichage du tableau. Avec cela sont créés les tableaux de commandes pour l'imprimante. La première séquence de commandes est PRINT. Elle fixe le pas de l'imprimante à 7/72 et envoie un code d'avance du papier. Pour envoyer cette séquence de commandes à l'imprimante, l'adresse du premier octet de la liste est chargée dans HL et TABOUT est appelée. TABOUT reconnaît la fin d'une séquence à l'octet nul.

Autrement, il y a aussi les séquences PRLIIN et PRREIN. PRLIIN (printer line init) envoie la commande qui fixe l'imprimante en mode point par point pour 639 octets (c'est à dire une ligne). PRREIN (printer re-init) remet l'imprimante en mode normal.

Afin que vous puissiez adapter le programme de HARDCOPY à d'autres imprimantes, nous avons laissé quatre octets libres après chaque séquence. Si votre imprimante ne doit faire aucune avance de papier, changez le code 24 de PRLIIN en 10 (LF).

La routine PRINT permet l'impression sur l'imprimante. D'abord, le compteur PRTCOU est décrémenté et la boucle est reprise lorsque le compteur a la valeur 0. Avec cela, il n'y a que 639 octets par ligne d'envoyés et non 640. Puis MCPRBU teste si l'imprimant est 'busy' (occupée). Si oui, le test est repris et quand la réponse est négative, l'octet qui se trouve dans l'accumulateur est envoyé vers l'imprimante avec MCPRCH.

```
.COPYRIGHT 1985 MICRO-APPLICATION.
.DAMS *.*.

                 ; Hardcopy par H.D. 18/10/85
                 ;   pour DMP 2000,
                 ; Epson FX80 et compatibles.
                 ; Fonctionne sur CPC 464,664 et 6128
                 ; sans adaptation
                 ;
                        ORG   #a000
                 scgelo EQU   #bc0b          ;Screen Get Location
                 scneli EQU   #bc26          ;Screeen next Line
                 scneby EQU   #bc20          ;Screen next Byte
                 mcprch EQU   #bd2b          ;MC print Character
                 mcprbu EQU   #bd2e          ;MC Pinter Busy ?
                 ;
#A000 21A8A0            LD    hl,prinit      ;Initialisation de
#A003 CD87A0            CALL  tabout         ; l'imprimante
#A006 CD0BBC            CALL  scgelo
#A009 57                LD    d,a
#A00A 1E00              LD    e,0
#A00C 19                ADD   hl,de          ;Adresse du point
                                             en haut
#A00D 22D5A0            LD    (tabend),hl    ; a gauche
#A010 1E1D              LD    e,29           ;Compteur de lignes
#A012 3E07              LD    a,7
#A014 32C4A0            LD    (bitanz),a     ;Bits par ligne
                                             d'impression
#A017 D5         neline PUSH  de             ;Sauvegarde du
                                             compteur de lignes
#A018 0608              LD    b,8            ;Modification de 8
                                             adresses de ligne
#A01A 7B                LD    a,e            ;Si ce n'est pas
#A01B FE01              CP    1              ; la derniere ligne
#A01D 200B              JR    nz,ok          ; tout est ok
#A01F 3E04              LD    a,4            ; sinon 4 bits par
#A021 32C4A0            LD    (bitanz),a     ; ligne imprimee
#A024 11CDA0            LD    de,tabmit      ; et remplissage du
```

```
                                            tableau
#A027 0604              LD   b,4            ; avec seulement 4
                                            elements
#A029 21                DEFB #21            ;Evite un JR
#A02A 11C7A0    ok      LD   de,tabanf
#A02D 2AD5A0            LD   hl,(tabend)    ;HL est le nouveau
#A030 EB        neli1   EX   de,hl          ;Chargement dans la
                                            table
#A031 73                LD   (hl),e
#A032 23                INC  hl
#A033 72                LD   (hl),d
#A034 23                INC  hl
#A035 EB                EX   de,hl
#A036 CD26BC            CALL scneli         ;Chargement et
                                            sauvegarde
#A039 10F5              DJNZ neli1          ; de l'adresse de
                                            ligne suivante
#A03B 21B0A0            LD   hl,prliin      ;Mode bitmap
#A03E CD87A0            CALL tabout         ; en marche
#A041 217F02            LD   hl,#27f        ; mais on n'envoie
#A044 22C5A0            LD   (prtcou),hl    ; que 639 octets
#A047 015008            LD   bc,#850        ;C=compteur
                                            d'octets=80
#A04A C5        neby    PUSH bc
#A04B 3AC4A0            LD   a,(bitanz)     ;Nombre des
#A04E 47                LD   b,a            ; bits imprimes
#A04F 97                SUB  a              ;Mise a 0 de l'accu
#A050 21D4A0            LD   hl,tablet      ;On commence avec
#A053 56        nebit   LD   d,(hl)         ; le dernier
                                            element de la table
#A054 2B                DEC  hl
#A055 5E                LD   e,(hl)         ;Lecture de
                                            l'adresse
#A056 2B                DEC  hl             ; de l'octet
#A057 EB                EX   de,hl
#A058 CB06              RLC  (hl)           ;Rotation de l'octet
#A05A 1F                RRA                 ;Rotation du carry
                                            dans l'accu
#A05B EB                EX   de,hl
#A05C 10F5              DJNZ nebit          ;Bit suivant
```

```
#A05E 1F                RRA               ;Ne pas utiliser le
                                          bit 7
#A05F CD93A0            CALL print        ;Impression
#A062 C1                POP  bc
#A063 10E5              DJNZ neby         ;Incrementer les 7
                                          elements de
#A065 21C7A0            LD   hl,tabanf    ; la table d'un
                                          octet
#A068 0607              LD   b,7          ;Vers la droite
#A06A 5E        neby1   LD   e,(hl)       ; 8 fois
#A06B 23                INC  hl           ; par octet
#A06C 56                LD   d,(hl)
#A06D 2B                DEC  hl
#A06E EB                EX   de,hl
#A06F CD20BC            CALL scneby
#A072 EB                EX   de,hl
#A073 73                LD   (hl),e
#A074 23                INC  hl
#A075 72                LD   (hl),d
#A076 23                INC  hl
#A077 10F1              DJNZ neby1
#A079 0608              LD   b,8
#A07B 0D                DEC  c            ;Pas encore la fin
                                          de la ligne?
#A07C 20CC              JR   nz,neby      ;Alors octet suivant
#A07E D1                POP  de
#A07F 1D                DEC  e            ;Derniere ligne?
#A080 2095              JR   nz,neline    ; non, alors
                                          reinitialiser
#A082 21BCA0            LD   hl,prrein    ; l'imprimante
#A085 1800              JR   tabout       ; puis fin
                ;
#A087 7E        tabout  LD   a,(hl)       ;Affichage
#A088 FE00              CP   0            ; de la table
#A08A C8                RET  z            ; jusqu'a
#A08B 23                INC  hl           ; l'octet nul
#A08C E5                PUSH hl           ;
#A08D CD93A0            CALL print
#A090 E1                POP  hl
#A091 18F4              JR   tabout       ;Octet suivant
```

```
                  ;
#A093 2AC5A0      print   LD   hl,(prtcou) ;Decrementer
#A096 2B                  DEC  hl          ; le compteur
                                           d'octet
#A097 22C5A0              LD   (prtcou),hl ; et le sauvegarder
                                           a nouveau
#A09A 4F                  LD   c,a         ;Memorisation
                                           temporaire du
#A09B 7C                  LD   a,h         ; caractere, test si
#A09C B5                  OR   l           ; le compteur
                                           d'octet
#A09D C8                  RET  z           ; est egal a zero
#A09E 79                  LD   a,c         ;Non, alors
                                           affichage de l'accu
#A09F CD2EBD      wait    CALL mcprbu      ;Printer Busy ??
#A0A2 38FB                JR   c,wait      ; oui, alors attente
#A0A4 CD2BBD              CALL mcprch      ;Impression du
                                           caractere
#A0A7 C9                  RET
                  ;
#A0A8 1B          prinit  DEFB 27          ;ESC
#A0A9 31                  DEFB "1"         ;Avance de 7/72
                                           pouce
#A0AA 0D                  DEFB #0d         ;CR et octet 0
#A0AB 00                  DEFB 0
#A0AC 0000                DEFW 0           ;Place pour
#A0AE 0000                DEFW 0           ; des modifications
                  ;
#A0B0 0D          prliin  DEFB 13
#A0B1 18                  DEFB 24          ;CANcel, ou LF
#A0B2 1B                  DEFB 27
#A0B3 2A                  DEFB "*"         ;Mode bitmap
#A0B4 01                  DEFB 1           ;Mode 1
#A0B5 7F02                DEFW #027f
#A0B7 00                  DEFB 0           ;Octet 0
#A0B8 0000                DEFW 0           ;Place pour
#A0BA 0000                DEFW 0           ; des modifications
                  ;
#A0BC 1B          prrein  DEFB 27          ;ESC
#A0BD 40                  DEFB "@"         ;Normalisation
```

```
#A0BE OD                    DEFB #0d        ;CR
#A0BF 00                    DEFB 0
#A0C0 0000                  DEFW 0          ;Place pour des
#A0C2 0000                  DEFW 0          ; modifications
                  ;
#A0C4 07          bitanz    DEFB 7
#A0C5 7F02        prtcou    DEFW #27f
                  tabanf    DEFS 6
                  tabmit    DEFS 7
                  tablet    DEFS 1
                  tabend    DEFS 2
                            END

Pass 2: 0 Errors

End of code:#A0D7
```

```
10 REM Hardcopy pour tous les CPC
20 REM appel avec call &a000 en MODE 2
30 FOR i=&A000 TO &A0C6
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 19650 THEN PRINT"erreur dans les  Datas":END
70 PRINT"ok!":END
80 DATA 21,A8,A0,CD,87,A0,CD,0B
90 DATA BC,57,1E,00,19,22,D5,A0
100 DATA 1E,1D,3E,07,32,C4,A0,D5
110 DATA 06,08,7B,FE,01,20,0B,3E
120 DATA 04,32,C4,A0,11,CD,A0,06
130 DATA 04,21,11,C7,A0,2A,D5,A0
140 DATA EB,73,23,72,23,EB,CD,26
150 DATA BC,10,F5,21,B0,A0,CD,87
160 DATA A0,21,7F,02,22,C5,A0,01
170 DATA 50,08,C5,3A,C4,A0,47,97
180 DATA 21,D4,A0,56,2B,5E,2B,EB
190 DATA CB,06,1F,EB,10,F5,1F,CD
200 DATA 93,A0,C1,10,E5,21,C7,A0
210 DATA 06,07,5E,23,56,2B,EB,CD
220 DATA 20,BC,EB,73,23,72,23,10
230 DATA F1,06,08,0D,20,CC,D1,1D
240 DATA 20,95,21,BC,A0,18,00,7E
250 DATA FE,00,C8,23,E5,CD,93,A0
260 DATA E1,18,F4,2A,C5,A0,2B,22
270 DATA C5,A0,4F,7C,B5,C8,79,CD
280 DATA 2E,BD,38,FB,CD,2B,BD,C9
290 DATA 1B,31,0D,00,00,00,00,00
300 DATA 0D,18,1B,2A,01,7F,02,00
310 DATA 00,00,00,00,1B,40,0D,00
320 DATA 00,00,00,00,07,7F,02
```

XII. LE BON TIMER DU CPC

Afin que vous n'oubliez pas le temps en programmant, nous avons dévelopé une horloge logicielle. Elle vous permettra d'éviter des ennuis avec les amis et la famille et elle constitue une extension intéressante pour certains programmes.

Les programmes tels l'horloge logicielle ne peuvent être écrits qu'avec l'aide des interruptions. La programmation des interruptions est encore plus difficile que la programmation normale en langage machine. Il est déjà assez difficile de tester un programme en langage machine. Mais pour les programmes qui utilisent des interruptions, c'est presque impossible, de les tester réellement car ils dépendent du temps.

Mais la programmation des interruptions ouvre des perspectives extraordinaires. D'abord, l'ordinateur ne fonctionnerait pas sans interruption. Un des travail des interruptions est par exemple le test du clavier. Naturellement, les interruptions BASIC sont traitées de manière interne.

Commençons donc. Dans tous les ordinateurs se trouve un quartz artificiel qui commande et synchronise le fonctionnement global. Ce quartz est donc l'horloge interne de l'ordinateur. Dans beaucoup d'ordinateurs, il y a plusieurs de ces timers. Avec un de ces timers, la broche de demande d'interruption (IRQ) du Z80 est mise à zero régulièrement (dans le CPC, cela se fait avec le gate array). Le programme qui fonctionne pendant ce temps est branché à une adresse indépendante du mode d'interruption.

Le Z80 est géré par le mode d'interruption 1. L'IRQ exécute un ordre RST &38 ou CALL &0038. L'IRQ est effectué dans le CPC 300 fois par seconde. Si l'interruption n'est pas inhibée par DI, l'adresse courante du programme est mémorisée sur la pile et un branchement est effectué vers l'adresse &38. Là-bas s'effectue un branchement à l'adresse &b989 (6128: &b941/ 664: &b941), c'est-à-dire dans la RAM où commence la routine d'interruption.

A partir de là, la ROM est sélectionnée et un branchement conditionnel est effectué à la routine de gestion des interruptions en &00b1. A cette adresse ne sont permises par exemple que les tests du clavier et l'incrément de la variable TIME.

Une deuxième routine ROM sert à gérer les interruptions BASIC. De plus toute la gestion du CHIP sonore est faite par interruptions.

Pour ajouter sa propre interruption à celles déjà existantes, il faut inhiber le branchement à la routine d'interruptions. A la fin de sa routine, il faut se brancher de nouveau sur la routine d'interruptions, c'est-à-dire à l'adresse &00b1.

Avec ce processus, nous pouvons développer nos propres routines sans limitation et de façon indépendante. A partir du moment où elles sont reliées à la routine d'interruption, elles sont appelées tous les 1/300 ième de seconde sans influer sur le fonctionnement de l'ordinateur.

Cela signifie que dans notre exemple d'horloge, celle-ci fonctionne sans qu'il soit besoin de s'en occuper. Tant que l'ordinateur est en route, l'heure est affichée à l'écran, à la position désirée. Nous allons maintenant expliquer les routines utiles.

Au début du programme, il y a une petite routine d'initialisation, qui relie l'ordre CALL pour la routine d'interruption avec notre propre routine. Les instructions suivantes transforment la routine d'initialisation de telle façon qu'elle relie la routine de nouveau lors d'un nouvel appel.

La routine commence au label START. D'abord, l'adresse de branchement en retour de la routine d'interruption est sauvegardée sur la pile. c'est-à-dire que notre propre routine peut ensuite être terminée simplement par RET et appelée à l'adresse &00b1.

Le premier compteur COUNT est diminué de un à chaque appel. Ainsi grâce à lui, le temps n'est affiché qu'une fois par seconde, donc toutes les 300 fois.

Il est intéressant de perdre le moins de temps possible lors de ce processus car autrement, l'ordinateur serait considérablement ralenti. Si le compteur COUNT est égal à zéro, il prend de nouveau la valeur 300 pour la prochaine boucle. Enfin, le programme d'horloge commence.

Pour sauvegarder le temps, on utilise 3 octets. Un pour l'heure, un pour les minutes et le dernier pour les secondes. Par exemple, on charge un 16 pour 16 heures dans l'octet des heures. La valeur de chaque octet est sauvegardée dans le format BCD. Cela signifie que chaque chiffre de la valeur décimale est sauvegardé sur 4 bits. BCD signifie décimal en code binaire. En raison de la spécificité du système hexadécimal, on peut aussi écrire ce qui suit: le nombre décimal &16 est 16 en BCD. Le format BCD n'est valable dans notre cas que pour des raisons de rapidité. Le calcul avec un chiffre d'un octet en BCD est à peine plus lent que le calcul avec des chiffres normaux, car le Z80 possède l'instruction DAA spécialement prévue pour le calcul en BCD. L'affichage d'un chiffre BCD est beaucoup plus simple et plus rapide que celui d'un chiffre normal. De cette manière, on perd le moins possible de temps de traitement.

En ce qui concerne les interruptions, il faut insister particulièrement sur le facteur temps.

Pour augmenter l'heure d'une seconde, l'adresse de l'octet des secondes est chargée dans le registre HL et le nombre maximum de secondes (60) est fixé dans le sous programme ZEIT.

Le sous programme ZEIT augmente le chiffre BCD se trouvant à l'adresse indiquée de un et teste si la valeur maximale a été atteinte. Si c'est le cas, le CARRY-FLAG est effacé, ou bien rétabli. Si la valeur n'a pas été atteinte, on effectue un saut à la routine principale pour afficher le temps. C'est possible car les minutes et les heures ne changent que lorsque 60 secondes se sont écoulées.

Lorsque la valeur maximale a été atteinte, ZEIT donne à l'octet la valeur zéro puis il le remet à zéro après 59 secondes ou minutes, etc...jusqu'à 24 heures.

Lorsque la valeur maximale des secondes est atteinte, HL prend la valeur de l'adresse des minutes et pour incrémenter celles-ci, ZEIT est appelé de nouveau. Lorsque la valeur maximale des minutes a été atteinte, les heures sont incrémentées, b ayant pris la valeur 24. Plus loin, les 3 octets sont fixés sur l'heure courante et peuvent être affichés.

Pour cela la mémoire POS prend pour valeur la position d'écran pour l'affichage. Puis la routine BCBYOU (BCD BYte OUt) permet de séparer les heures, les minutes et les secondes par deux points. L'accumulateur prend la valeur &30, le code ASCII de '0'. Puis un chiffre BCD sous 4 bits subit une rotation avec RLD, à condition que HL pointe sur l'octet voulu. Par cela, le code ASCII de chiffre considéré se trouve dans l'accumulateur et peut être affiché. L'affichage est produit par l'appel du sous programme PRINT, qui calcule la position pour chaque affichage et la mémorise. Voici maintenant le listing assembleur complet :

```
.COPYRIGHT 1985 MICRO-APPLICATION.
.DAMS *.*.

                   ;
                   ; Logiciel d'horloge pour tous les AMSTRAD
                   ; H.D. 22/9/85
                   ;
                           ORG   #a000
                   ;
                   ; Routine d'initialisation
                   ;
#A000 210DA0               LD    hl,start      ;Routine
                                               d'interruptions
#A003 2251B9               LD    (#b951),hl    ;Patcher
                   ; Pour 464: ld (#b949),hl
#A006 21B100               LD    hl,#b1
#A009 2201A0               LD    (#a001),hl
#A00C C9                   RET
                   write   EQU   #bdd3         ;(464, 664 et 6128)
                   intcou  EQU   300
                   positi  EQU   #4700         ;Mode 2
                   doppun  EQU   #3a
                   ;
#A00D 21B100       start   LD    hl,#b1        ;Adresse de retour
#A010 E5                   PUSH  hl            ;Routine de service
                                               des interruptions
#A011 2A7EA0               LD    hl,(count)    ;Aller chercher
                                               compteur
#A014 2B                   DEC   hl            ;Diminuer pour que
                                               le "reste"
#A015 227EA0               LD    (count),hl
#A018 7C                   LD    a,h           ; ne soit sorti que
                                               toutes les secondes
#A019 B5                   OR    l             ;Donc compteur
                                               different de
#A01A C0                   RET   nz            ; zero, alors
                                               termine
#A01B 212C01               LD    hl,intcou     ;Initialiser a
```

```
                                                      nouveau
#A01E 227EA0              LD    (count),hl           ; le compteur
                 ;
#A021 2182A0              LD    hl,sec               ;Adresse de l'octet
                                                      seconde
#A024 0660                LD    b,#60                ;60 secondes BCD
                                                      maximum
#A026 CD52A0              CALL zeit                  ;Incrementer les
                                                      secondes
#A029 380A                JR    c,anzei              ;Pas encore 60,
                                                      alors afficher
#A02B CD52A0              CALL zeit                  ;Incrementer les
                                                      minutes
#A02E 3805                JR    c,anzei
#A030 0624                LD    b,#24                ;24 heures maximum
#A032 CD52A0              CALL zeit                  ;
#A035 210047     anzei    LD    hl,positi            ;Position pour la
                                                      sortie
#A038 227CA0              LD    (pos),hl
#A03B 2180A0              LD    hl,stund
#A03E CD5DA0              CALL bcbyou                ;Sortie d'un octet
                                                      en format BCD
#A041 3E3A                LD    a,doppun
#A043 CD6DA0              CALL print
#A046 CD5DA0              CALL bcbyou                ;Sortir les minutes
#A049 3E3A                LD    a,doppun
#A04B CD6DA0              CALL print
#A04E CD5DA0              CALL bcbyou                ;Sortir les secondes
#A051 C9                  RET
                 ;
                 ; Sous programme d'incrementation des
                                                      compteurs horaires
                 ;
#A052 7E         zeit     LD    a,(hl)               ;Ancienne heure
#A053 C601                ADD   a,1                  ;Incrementer
#A055 27                  DAA                        ;Les octets sont en
                                                      format BCD
#A056 77                  LD    (hl),a               ;Sauvegarder
#A057 B8                  CP    b                    ;Maximum atteint ?
#A058 D8                  RET   c                    ;Non, alors termine
```

```
#A059 97                SUB  a            ;Oui alors continuer
#A05A 77                LD   (hl),a       ; avec "0"
#A05B 2B                DEC  hl           ;Prochaine fois
                                          minutes (heure)
#A05C C9                RET
                ;
                ; Routine de sortie d'un octet BCD
                ;
#A05D 3E30      bcbyou  LD   a,#30        ;Sortie de nombres
                                          (asc("0")=#30)
#A05F ED6F              RLD               ;Faire subir une
                                          rotation au chiffre
#A061 CD6DA0            CALL print        ; de plus grande
                                          valeur
#A064 ED6F              RLD
#A066 CD6DA0            CALL print        ;Chiffre de moindre
                                          valeur
#A069 ED6F              RLD               ;Restaurer valeur
#A06B 23                INC  hl           ;Prochaine fois
                                          minutes (secondes)
#A06C C9                RET
                ;
#A06D E5        print   PUSH hl
#A06E F5                PUSH af
#A06F 2A7CA0            LD   hl,(pos)
#A072 24                INC  h
#A073 227CA0            LD   (pos),hl
#A076 CDD3BD            CALL write
#A079 F1                POP  af
#A07A E1                POP  hl
#A07B C9                RET
                ;
                pos     DEFS 2
#A07E 2C01      count   DEFW 300
#A080 00        stund   DEFB 0
#A081 00                DEFB 0
#A082 00        sec     DEFB 0
                        END

Pass 2: 0 Errors
```

End of code:#A083

```
10 REM chargeur basic pour le programme HORLOGE
20 REM Initialisation avec Call &a000
30 FOR i=&A000 TO &A07F
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 14784 THEN PRINT"Erreurs dans les Datas":END
70 ' pour 464:IF s<> 14776 THEN .........
80 PRINT"ok!":END
90 DATA 21,0D,A0,22,51,B9,21,B1
100 '464:..,..,..,..,49,..,..,..
110 DATA 00,22,01,A0,C9,21,B1,00
120 DATA E5,2A,7E,A0,2B,22,7E,A0
130 DATA 7C,B5,C0,21,2C,01,22,7E
140 DATA A0,21,82,A0,06,60,CD,52
150 DATA A0,38,0A,CD,52,A0,38,05
160 DATA 06,24,CD,52,A0,21,00,47
170 DATA 22,7C,A0,21,80,A0,CD,5D
180 DATA A0,3E,3A,CD,6D,A0,CD,5D
190 DATA A0,3E,3A,CD,6D,A0,CD,5D
200 DATA A0,C9,7E,C6,01,27,77,B8
210 DATA D8,97,77,2B,C9,3E,30,ED
220 DATA 6F,CD,6D,A0,ED,6F,CD,6D
230 DATA A0,ED,6F,23,C9,E5,F5,2A
240 DATA 7C,A0,24,22,7C,A0,CD,D3
250 DATA BD,F1,E1,C9,00,00,2C,01
```

Au cas où vous n'auriez pas d'assembleur, vous trouverez également un chargeur BASIC.

Autrement, voici un petit programme en BASIC avec lequel vous pourrez afficher l'heure de façon simple. Si vous voulez arrêter l'horloge, c'est-à-dire stopper la routine d'interruptions, il suffit d'entrer de nouveau CALL &A000.

Pour la remettre en route, il faut exécuter :

```
POKE &A001,&D:POKE &A002,&A0
```

puis le CALL.

```
10 REM affichage de l'horloge
20 MEMORY &9FFF
30 MODE 2
40 ' LOAD"uhr1.obj
50 LOCATE 15,7:PRINT"H O R L O G E"
60 LOCATE 1,11
70 INPUT"affichage sur 12 ou 24 heures (12/24) ?";e
80 PRINT
90 IF e<>12 AND e<>24 THEN 70
100 POKE &A031,VAL("&"+STR$(e))
110 zeibas=&A080
120 INPUT"heure (hh,mm,ss)";h,m,s
130 PRINT
140 POKE zeibas,VAL("&"+STR$(h)):POKE zeibas+1,VAL("&"+STR$(
m)):POKE zeibas+2,VAL("&"+STR$(s))
150 INPUT "Position (colonne,ligne)";sp,ze
160 IF sp<1 OR ze<1 THEN 150
170 IF sp=1 THEN sp=257
180 POKE &A036,ze-1:POKE &A037,sp-2
190 ' call &a000
```

TROISIEME PARTIE

TRUCS ET ASTUCES EN LANGAGE MACHINE

XIII. PROGRAMMATION EN LANGAGE MACHINE

Lorsque la rapidité est un facteur important, le programmeur est toujours amené à programmer directement ou indirectement en langage machine. Directement signifie que le code machine est introduit directement dans l'ordinateur. Pour cela il existe un grand nombre d'assembleurs qui facilitent ce travail. Indirectement signifie que l'on utilise un compilateur. Si on se sert par exemple d'un compilateur BASIC, le programme est tout d'abord écrit en BASIC comme d'habitude. Le compilateur traduit alors ce BASIC en code machine qui peut être utilisé sans le code objet.

Pour jeter un oeil sur l'intérieur de la machine, nous allons faire quelques petites routines directement en assembleur et comparer leur gain en rapidité par rapport à un langage interprété, tel le BASIC du CPC. Pour cela nous n'avons pas besoin d'un assembleur. Des petits chargeurs BASIC nous permettront de mémoriser le code machine.

Démarrez le petit programme en BASIC qui suit :

```
50 mode 2
100 for adresse=49152 to 65535
110 poke adresse,255
120 next
```

Description du programme :

Les 16K RAM haut de la mémoire du CPC (&C000-&FFFF) sont réservés à l'écran. Les contenus numériques des mémoires représentent les points qui apparaissent à l'écran pour former des images. Avec notre exemple de programme, nous affichons tous les points de l'écran.

50: l'instruction mode 2 efface l'écran et fixe le curseur dans le coin gauche en haut de l'écran.

100: La boucle FOR-NEXT commence avec l'adresse de début de l'écran graphique et se termine avec l'adresse de fin.

110: La valeur 255 est introduite dans toutes les cases mémoires.

Le déroulement d'un tel programme prend environ 40 secondes. Pour 16384 cases à visiter, cela signifie que la transformation de chaque case dure environ 2,4 millisecondes. Pour faire une comparaison avec un programme en langage machine, qui est construit à peu près comme le programme en BASIC, exécutez le programme suivant :

```
10 mode 2:somme=0
20 for i=40960 to 40979
30 read a$:valeur=val('&'+a$)
40 poke i,valeur:somme=somme+valeur:next
50 if somme<>1531 then print 'erreur dans les data!':end
60 print 'tapez une touche pour commencer la routine en langage
machine'
70 call &bb06:call 40960
80 end
90 data 21,00,40,01,01,00,11,00,c0,3e
100 data ff,12,13,ed,42,c2,0b,a0,c9,00
```

Avec cette routine en assembleur, on réduit le temps total à .2 seconde, soit 12 microsecondes par case mémoire. Bien que cette routine ne soit pas la solution la plus rapide, elle vous montre les possibilités de l'assembleur.

Explication du programme :

Le programme en langage machine est constitué de valeurs hexadécimales qui sont dans les lignes de datas 90 et 100. Dans la boucle FOR-NEXT (ligne 20-40), les valeurs sont lues, transformées dans une valeur numérique puis chargées dans les mémoires allant de 40960 à 40979. L'instruction CALL &BB06 en ligne 70 appelle une routine du système d'exploitation qui teste le clavier jusqu'à ce qu'une touche soit pressée. Puis l'instruction suivante est exécutée. Dans notre cas, il s'agit de CALL 40960 qui appelle la routine écrite en mémoire dans la boucle FOR-NEXT. A la fin l'ordinateur revient au sommet de la mémoire BASIC et affiche READY après être revenu en ligne 80 (END).

Que signifie exactement les valeurs hexadécimales des lignes data ?

Pour donner une réponse compréhensible à cette question, il faut s'intéresser de plus près au Z80.

Après une introduction au langage du CPU (trucs et astuces n01), il faut maintenant se pencher un peu mieux sur la boite à astuces de l'ordinateur. Nous allons présenter des instructions importantes, utiles et rapides du Z80 et les expliquer avec des routines pratiques.

Pour les comprendre, il faut d'abord bien connaître les registres, leur structure, leur fonctionnement et leur efficience.

13.1. Les Registres du Z80

Les registres sont des cases mémoires dans le CPU. Bien que le Z80 soit un microprocesseur 8 bits, il possède des registres 16 bits. Les registres sont désignés comme suit :

registres 8 bits: A,B,C,D,E,F,H,L
registres 16 bits: BC,DE,HL,SP,PC

Vous n'avez là que les registres les plus importants. Nous allons ignorer les autres car ils ne sont pas utiles pour la compréhension du système. J'aimerais aussi rassurer les lecteurs qui ont remarqué que certains des registres 16 bits ont des désignations qui sont des combinaisons des registres 8 bits. Cela a des raisons logiques...Mais nous en parlerons plus tard.

13.1.1. Registres 8 bits

Les registres 8 bits que nous avons nommés ne sont pas équivalents. Les registres A et F ont une position spéciale par rapport aux autres et nous devrions les présenter ainsi:

A,F,   B,C,D,E,H,L

LE REGISTRE A (ACCUMULATEUR)

Toutes les opérations logiques et arithmétiques sur 8 bits utilisent le registre A. Le résultat de ces opérations se trouve alors dans le registre A alors que les autres registres ne sont pas modifiés.

1. Exemple de l'addition de deux nombres

```
LD A,6   charge 6 dans l'accumulateur
LD D,3   charge le registre D avec 3
ADD A,D  additionne le contenu de d à l'accumulateur
```

Après l'opération ADD A,D, A a la valeur 9, alors que D garde la valeur 6.

2. Exemple d'une soustraction de deux nombres

```
LD A,&54   charge l'accumulateur avec &54
LD L,&54   charge le registre L avec &54
SUB A,L    Soustrait le contenu de d de l'accumulateur
```

Après cette soustraction, l'accumulateur a la valeur 0.

Ces opérations arithmétiques ne modifient pas seulement le contenu de l'accumulateur. Il y a aussi une modification du registre FLAG. Celui-ci donne des informations sur certaines modifications qui peuvent survenir lors des opérations avec l'accumulateur. Pour expliquer les modifications, certains bits du registre F sont effacés ou fixés. Il n'est possible de réaliser certaines opérations qu'avec les informations que donne le registre FLAG. Voici maintenant la description de ce registre.

LE REGISTRE F (REGISTRE FLAG)

Comme vous le savez déjà, on peut obtenir certaines informations sur le résultat d'opérations arithmétiques et logiques avec ce registre. Le flag zéro n'est qu'un des six flags de ce registre, mais c'est sans doute le plus important avec le carry (c).

Structure du registre FLAG :

Dans ce qui suit, vous voyez la position des flags dans le registre flag.

FLAG	s	z		h		p/v	n	c
bit n0	7	6	5	4	3	2	1	0

Description des flags :

S : sign — 0 résultat positif
1 résultat négatif

Z : zéro — 0 résultat différent de 0
1 résultat égal à 0

H : halfcarry 0 aucun déplacement du 3 au bit 4 de l'accu
1 déplacement du bit 3 vers le bit 4

P/V : Parity/overflow 0 parité impaire pas de déplacement du bit 6 vers le bit 7
1 parité paire déplacement du bit 6 vers le bit 7

N : flag de soustraction 0 après une addition
1 après une soustraction

C : CARRY 0 aucun déplacement du bit 7 vers le bit 8
1 déplacement du bit 7 vers le bit 8

LE BIT SIGN (S)

Avec le 8ème bit d'un octet, on peut obtenir 2^8=256 valeurs, de 0 à 255. Dans la représentation signée, ce sont les valeurs de -128 à 127. Dans ce cas, les valeurs sont inscrites dans les bits de 0 à 6 et le signe est fixé par le bit 7: Si le signe est positif, le bit 7 est effacé (0). Si le bit 7 est fixé (1), le signe est négatif.

Ainsi le nombre binaire 11111111 correspond au chiffre décimal -1.

LE BIT ZERO (Z)

Lorsqu'un octet est modifié par une opération logique ou arithmétique ou bien par une rotation ou un déplacement, le flag Z à 1 indique que l'octet est nul. Si le flag est 0, la valeur de l'octet est différente de 0. Lors de la comparaison de deux octets (p.ex. CP E), le bit Z est nul quand les octets ont la même valeur et il est à 1 si les valeurs diffèrent.

Ceci est simple à comprendre quand on sait que l'instruction CP S soustrait l'opérande S de l'accumulateur. Pour les trois fonctions suivantes du registre Z, j'aimerais introduire l'instruction de test BIT b,r. l'instruction BIT 5,D teste le cinquième bit du registre D. Si le contenu est 1, alors le flag Z est nul, si le contenu est nul, le flag Z prend la valeur 1.

LE BIT HALF CARRY (H)

Le flag H montre un déplacement du 'nibble' (bit 0-3) inférieur vers le nibble supérieur (bit 4-7) d'un octet. L'utilisation est valable pour l'arithmétique en BCD et la comparaison décimale (DAA). Si le nibble inférieur a une valeur plus grande que 9, il faut que se produise un déplacement vers le nibble supérieur, car un nibble représentant un nombre décimal ne doit pas être supérieur à 9. Ce déplacement est montré par le flag H.

LE BIT DE PARITE ET OVERFLOW (P/V)

Ce flag possède plusieurs fonctions :

a) Malgré la grande qualité des hardware actuels, il faut éviter les erreurs dans la transmission des données, mais ce n'est pas toujours gênant. La parité est souvent utilisée pour la transmission de caractères en ASCII (7 bits) en tant que huitième bit. Si le nombre des bits qui sont à 1 est pair, alors le bit de parité prend la valeur 1. Si ce nombre est différent de 1, le bit de parité prend la valeur 0. Si un seul bit est mal transmis, la parité n'est plus exacte et on retransmet l'octet.

b) Lorsque le résultat d'une addition ou d'une soustraction est plus grand que 127, le 7ème bit réservé pour le signe est modifié. Cela est indiqué par le flag P/V.

c) Lors d'un transfert de blocs ou d'instructions de recherche de bloc, ce flag est modifié en relation avec le registre BC. Ce registre a le rôle d'un compteur pour les instructions indiquées. Si le registre compteur BC est égal à 0, le flag P/V prend la valeur 0. Si BC est différent de 0, P/V est égal à 1.

d) Pour les instructions LD a,i et LD a,r, le flag P/V contient la valeur du flip-flop d'activation de l'interruption IFF2. Avec cela, il est possible de demander ou de mémoriser le contenu de IFF2.

LE BIT DE SOUSTRACTION (N)

Le programmeur n'utilisera habituellement pas ce flag. L'utilisation spécifique de ce flag est faite pour des comparaisons en décimal (DAA) après une addition ou une soustraction. Cette comparaison est menée différemment après une addition qu'après une soustraction. Le comportement du CPU devant le DAA est indiqué par le contenu du flag N.

LE BIT CARRY (C)

Le bit de retenue indique lors d'une addition ou d'une soustraction si ces opérations ont conduit à un dépassement de valeur possible.

Lors des rotations ou des déplacements, le bit carry est utilisé comme nouveau bit.

Le CARRY est effacé par les opérations logiques AND, OR et XOR. Ces instructions du Z80 efface le carry lorsque cela est nécessaire.

Pourquoi ces opérations logiques effacent-elles le carry ?

Voyons un exemple avec la valeur &D3 dans l'accu :

```
    11010011
AND 11010011
    --------
    11010011
```

Comme on ne peut pas dépasser la valeur avec cette instruction, pas plus qu'avec OR ou XOR ni quand on compare l'accu avec un autre registre, le carry-flag est à 0. Ces instructions effacent donc le carry.

QUELQUES INFORMATIONS SUR QUAND ET COMMENT LES FLAGS SONT FIXES, EFFACES ET UTILISES.

Au premier abord, il peut paraître incompréhensible que le flag-zero prenne la valeur 1 exactement quand le registre testé par une opération devient égal à 0, et vice-versa. Ce processus devient cependant plausible, lorsque l'on considère le contenu du flag comme un indicateur.

Un flag peut donc prendre deux valeurs, vrai ou faux. 1 signifie vrai et 0, faux. Le Z80 possède de nombreuses instructions de branchement. Le CPU sait si les conditions nécessaires à un branchement sont remplies grâce aux flags.

Par exemple, l'instruction de branchement JP C, adresse beaucoup utilisée doit être utilisée en relation avec le registre des flags. Lors de l'addition du registre A avec une autre valeur, le résultat peut être une valeur supérieure à &ff. Comme nous le savons, le CPU n'est pas gêné par le dépassement de l'accumulateur car le bit carry est considéré comme neuvième bit dont la valeur est 2^8 (256). Si ce cas survient, le programme doit se brancher sur une autre partie de programme. Ceci est possible avec l'instruction de branchement décrite précédemment JP C,adresse qui est active lorsque le bit carry est à 1.

Si nous voulons effectuer un branchement lorsque le bit C est 0, il faut utiliser l'instruction JP NC,adresse qui est activée lorsque le carry est nul.

LES REGISTRES B,C,D,E,H,L

Ces registres sont semblables. On peut leur donner directement une valeur, on peut leur donner la valeur d'autres registres, ce qui est inutile à mon avis, lorsque l'on ne veut pas ralentir le processus. Les registres peuvent prendre des valeurs en RAM et rendre ces valeurs. On peut relier les registres avec le contenu de l'accumulateur de façon logique ou arithmétique ou bien les modifier par rotation ou déplacement. Ce sont les utilisations principales de ces registres 8 bits.

13.1.2. LES REGISTRES 16 BITS BC,DE,HL,PC ET SP

Les registres 8 bits B et C, D et E ainsi que H et L peuvent être liés les uns aux autres pour obtenir des registres 16 bits avec des instructions spécifiques, et cela dans un microprocesseur 8 bits.

Cela a naturellement des avantages considérables. Comment adresser la 40000ième place mémoire parmi les 65536 disponibles ? Avec des mots de 8 bits, on peut établir 256 valeurs ou adresses différentes. Mais avec deux octets constituant un registre 16 bits, on peut adresser 2^16=65536 cases mémoires différentes.

Les trois registres 16 bits disposent de fonctions spécifiques. Bien que leurs structures ne soient pas différentes, le processeur prend les paramètres nécessaires pour certaines opérations dans un registre ou dans l'autre.

Ainsi le registre HL peut mémoriser l'accumulateur sur 16 bits. On dispose de trois opérations arithmétiques différentes avec HL, DE et BC.

Le registre BC est souvent utilisé comme compteur. Cela se fait avec les instructions de transfert de blocs ou de recherche, dont nous parlerons mieux plus loin.

LE REGISTRE PC (PROGRAM COUNTER)

Afin d'éviter tout malentendu : le registre PC est un pur registre 16 bits. Mais on a une influence limitée sur ce registre.

La suite d'instructions d'un programme est toujours mémorisée dans une zone mémoire à l'extérieur du processeur. Dans le processeur lui-même on trouve seulement l'instruction en cours. Cette instruction est prise par le CPU à l'adresse indiquée par le PC. On peut donc comparer le PC à un pointeur. Il pointe sur l'adresse où se trouve l'Instruction ou la valeur suivante. Après que le processeur ait pris une instruction dans la mémoire, le PC est amené sur la position suivante. Le PC est augmenté d'une valeur qui est fonction de la longueur de l'instruction (il existe des instructions qui font 1, 2, 3 ou 4 octets.).

LE REGISTRE SP (STACK POINTER)

Le stack pointer a un rôle de pointeur. Le contenu du SP est l'adresse de l'élément supérieur de la pile (stack).

Mais qu'est-ce qu'un stack (ou une pile) ?

Alors que l'on peut adresser à n'importe quel moment n'importe quelle case mémoire, on n'a accès qu'à l'élément supérieur de la pile. On peut comparer cela à une pile de livres. Si on a mis plusieurs livres dans un carton, on ne peut prendre que le livre du dessus. Si on veut saisir celui du dessous, il faut d'abord enlever tous les autres. Cette méthode de saisie s'appelle LIFO. LIFO signifie LAST IN FIRST OUT, c'est-à-dire, le dernier élément entré sera le premier sorti. L'exemple donné avec la pile de livres n'a qu'un défaut. La pile de l'ordinateur est construite du bas vers le haut.

Le premier entré se trouve à la plus haute place dans la pile, ainsi les valeurs entrées par la suite auront des adresses mémoire plus faibles. Cela n'a cependant pas grande importance car la gestion est prise en charge par le CPU. Il faut quand même le savoir si on veut déplacer la pile.

Cela n'est pas fréquent car il y a toujours une place réservée en RAM pour la pile. Mais il ne faut jamais y entrer d'autres valeurs. Cela conduirait automatiquement au 'plantage' du système.

Je vous rappelle que le SP pointe toujours sur l'élément le plus haut de la pile, c'est-à-dire le dernier entré.

Nous avons donc décrit les registres les plus importants du CPU Z80. Il existe encore quelques registres dont je ne vais pas encore parler car ils compliqueraient l'explication et ne sont pas utiles pour avoir une compréhension globale du système.

Maintenant que nous comprenons le Z80 en gros, voyons les instructions de ce microprocesseur. Il serait absurde de vouloir connaître toutes les instructions dès le départ (il y en a plus de 500), mais nous allons étudier celles qui pourraient nous servir pour résoudre plusieurs problèmes.

13.3. Un exemple détaillé de programmation en langage machine

Le premier exemple de programme que je vais vous présenter va être le programme pour effacer l'écran.

Le programme en assembleur est situé comme nous l'avons vu aux lignes 90 et 100 :

```
 90 data 21,00,40,01,01,00,11,00,c0,3e
100 data ff,12,13,ed,42,c2,0b,a0,c9,00
```

Le programme BASIC sert à inscrire ces valeurs en mémoire à partir de 40960 avec des pokes.

Ces lignes de data n'ont une signification que pour les professionnels de l'assemblage. Maintenant nous allons écrire ce programme sous un aspect compréhensible. Cette forme est appelée listing de mnémoniques. Les mnémoniques sont des instructions réduites qui sont utilisées pour programmer en assembleur.

Lorsqu'on lit le code hexadécimal 3e ff, il faut une grande habitude pour savoir ce qu'il signifie. La représentation mnémonique de cette instruction est LD A,&FF. On comprend déjà plus facilement ce que cela veut dire.

LOAD	ACCU	with	HEX FF
charger	accu	avec	hex FF
LD	A	,	& FF

Le listing de programme qui suit est très clair. Dans la première colonne se trouve l'adresse mémoire, dans la seconde colonne, le code hexadécimal et dans la troisième colonne, l'instruction en assembleur. Puis le listing est expliqué par la colonne EXPLICATIONS.

ADRESSE	CODE HEX	MNEMONIC	REMARQUE
A000	21 00 40	LD HL,&4000	charge compteur
A003	01 01 00	LD BC,&0001	charge decrement
A006	11 00 C0	LD DE,&C000	première case écran
A009	3E FF	LD A,&FF	Accu prend valeur FF
A00B	12	LD (DE),A	mem De prend valeur Accu
A00C	13	INC DE	incremente DE de 1
A00D	ED 42	SBC HL,BC	soustrait BC de HL
A00F	C2 0B A0	JP NZ,&A00B	saut (&A00B si z=0
A012	C9	RET	retour au prg principal
A013	00	NOP	no operation

Les adresses de la colonne gauche n'indiquent que le premier octet de chaque ligne. Ainsi, pour les instructions de 3 octets, deux adresses sont sautées. On s'habitue vite à ce fait.

Les quatre premières instructions du listing sont connues. Elles chargent des valeurs dans les registres, leur abréviation est aisément compréhensible.

L'instruction LD (DE),A nous est encore inconnue.

Quand un nom de registre est entre parenthèses comme dans ce cas, le contenu du registre est considéré comme une adresse.

Elle contient à ce moment la valeur &C000 (voir ligne 3). Ainsi, l'instruction LD (DE),A charge le contenu de A dans la case mémoire &C000.

La deuxième instruction que nous ne connaissons pas encore est INC DE. INC signifie INCrémente. Cette instruction incrémente le contenu du registre donné de 1. Après cette instruction, le registre DE aura donc pour valeur &C001. Tous les registres dont nous vous avons parlé, excepté le program counter peuvent être incrémentés.

L'instruction SBC HL,BC est nouvelle elle aussi. SBC signifie soustraction avec carry (retenue). Le registre HL est diminué de la valeur de BC. Pour éviter des ennuis avec cette instruction, il faut toujours faire attention à ce que le bit carry soit à l'état désiré. Si on ne connait pas l'état du bit C, il vaut mieux l'effacer. Cela peut être fait avec les instructions AND A, XOR A et OR A. Dans notre exemple, j'ai tenu compte de ce fait pour la simplicité.

La dernière des nouvelles instructions que nous allons voir est un branchement conditionnel JP NZ,&A00B. (vous connaissez déjà RET et NOP de TRUCS ET ASTUCES tome I)

Cette instruction est toujours exécutée lorsque la condition nécessaire au branchement est remplie. Pour l'instruction JP NZ,adresse NZ signifie non zéro, c'est-à-dire différent de 0. Tant que le flag zero est égal à 0, l'instruction est exécutée. Dans cet exemple, cela se passe jusqu'à ce que le registre HL soit égal à 0, donc &4000 (16384) fois.

Cela correspond au nombre de cases mémoires de l'écran. Lorsque l'instruction SBC HL,BC met à zéro le registre HL, le flag zéro change d'état. La condition pour le branchement n'est plus remplie et il n'est plus exécuté. L'instruction suivante est RET. Elle est exécutée et l'ordinateur retourne an BASIC. Il est disponible et affiche READY.

UNE ETUDE DE RAPIDITE DE LA ROUTINE

Lorsque l'on choisit le langage machine pour des raisons de rapidité, il faut prêter une attention particulière au choix des instructions utilisées.

Dans cet esprit, la routine que nous venons de présenter perd de l'intérêt. Mais cela démontre que la solution directe parait souvent élégante mais qu'il existe d'autres solutions qui peuvent sembler bizarres ou bien trop longues.

La programmation en assembleur ne suit pas obligatoirement les conventions de programmation en langage évolué. Ainsi, l'effacement de l'écran peut être réalisé de façon plus efficace par un programme long que par un programme court et apparemment plus rapide. Bien entendu, il faut faire un compromis entre la rapidité et le temps de programmation.

Pour estimer la rapidité de notre routine, il serait absurde de nous tenir prêts à côté de l'ordinateur avec un chronomètre. Il est préférable de se servir d'une calculette.

Cela ne fonctionne évidemment que si l'on sait combien de temps le CPU met à exécuter les instructions. Dans l'exemple de notre routine, je vais vous donner les valeurs. Les chiffres que vous lirez après les mnémoniques sont les durées des instructions en microsecondes.

```
         LD HL,&4000    5
         LD BC,&0001    5
         LD DE,&C000    5
         LD A,&FF     3.5
-------------------------
boucle   LD (DE),A    6.5
         INC DE         3
         SBC HL,BC    7.5
         JP  NZ,boucle  5
-------------------------
         RET            5
```

Les quatre premières lignes du programme et la dernière ne seront exécutées qu'une seule fois, donc nous ne calculerons le temps d'exécution qu'une fois. La somme est 23.5*10^-6 secondes.

La partie de programme qui se trouve au label 'boucle' est une boucle qui est exécutée &4000 fois. Il faut donc multiplier la somme des durées des quatre lignes par &4000. Chaque somme est 22, donc le temps total est &4000*22*10^-6 secondes. Si on ajoute la durée des cinq autres lignes, le programme total dure 0.3604715 secondes.

Comme l'instruction de branchement travaille plus rapidement lorsque la condition est négative, il faudrait réduire un peu le temps total, mais cela ne fait que quelques fractions de microsecondes.

Nous allons maintenant accélérer la routine. Mais avant cela, il faut expliquer clairement ce qui se passe quand on augmente le contenu d'un registre. Lorsque la valeur maximale de ce registre est atteinte, ce registre retombe à zéro et l'incrémentation recommence. Nous voici au point où nous devons réfléchir comment accélérer notre routine. Lorsque le registre considéré est de nouveau à zéro, il a quitté la mémoire écran intéressante pour nous. Ce sera la condition de fin pour notre routine.

Jusqu'ici nous avons utilisé un registre de compteur (HL=&4000) outre notre registre d'adresses. Nous gagnerons le temps utilisé pour soustraire HL. Le registre d'adresses DE est incrémenté après la dernière adresse &FFFF et devient donc nul.
Après la rationalisation du travail de comptage, notre exemple devient :

```
        LD  DE,&C000   adresse de début
        LD   A,&FF     valeur à mémoriser
boucle  LD (DE),A      mémorisation
        INC DE         incrément de l'adresse
        JP  NZ,boucle  branchement conditionnel
        RET            retour au PRG principal
```

Cela semble être une solution pratique. Nous avons économisé trois lignes, deux registres doubles (HL et BC) et ce qui est très important, la ligne SBC HL,BC n'est plus exécutée. Cela gagne .12288 secondes c'est-à-dire 33% de la durée du programme.

Malheureusement, il y a un hic, que chaque programmeur en assembleur a rencontré au moins une fois.

L'instruction INC DE n'a aucun effet sur le registre de flag duquel l'instruction de branchement conditionnel tire sa condition. Même quand DE est décrémenté, le flag zéro ainsi que tous les autres demeurent inchangés. La condition de branchement n'est jamais remplie. D'abord, l'écran complet est rempli avec &FF. Puis le registre d'adresses DE prend la valeur zéro. Alors, on se retrouve en bas de la mémoire du CPC. Là se trouvent certaines routines indispensables du système d'exploitation qui ne doivent pas être effacées. L'ordinateur se plante...

Mais nos travaux de préparation n'étaient pas inutiles. Mais il faut faire attention d'une manière ou d'une autre à ce que le flag zéro soit fixé après INC DE en fonction du résultat de l'instruction INC. L'instruction BIT 7,D est utile pour cela.

J'explique son fonctionnement dans le listing qui suit:

```
        LD  DE,&C000
        LD  A,&FF
boucle  LD (DE),A
        INC DE
        BIT 7,D
        JP  NZ,boucle
        RET
```

Le registre d'adresses DE prend la valeur &C000 au début de la routine. L'écriture binaire permet de donner l'état de ce chiffre sur 16 bits :

registre	D		E	
HEXA	C	0	0	0
binaire	1100	0000	0000	0000
bit n0	7654	3210	7654	3210

Comme on le voit ici, les bits 6 et 7 du registre D sont fixés à 1 dès le début alors que les 14 bits suivants sont à zéro. Lors de l'incrémentation de DE, Les 14 bits nuls changent d'état. Lorsque DE a la valeur &FFFF (11111111 11111111 en binaire) et qu'il est incrémenté une nouvelle fois, la valeur est dépassée et les 16 bits du registre DE ainsi que les bits 6 et 7 de D redeviennent nuls.

A ce point, l'instruction BIT 7,D 'attend', ce qui modifie le flag zéro contrairement aux instructions d'incrémentation 16 bits. L'instruction fixe le flag Z à 1, lorsque le bit testé du registre devient nul. Dans ce cas particulier, on pourrait aussi utiliser l'instruction BIT 6,D, car le bit 6 du registre D est mis à 1 dès le début et change d'état avec le bit 7.

Les deux instructions sont absolument semblables.

La comparaison de la durée de cette routine avec la première que nous avons développée est encore plus intéressante.

```
          LD DE,&C000   5
          LD A,&FF      3.5
boucle    LD (DE),A     6.5
          INC DE        3
          BIT 7,D       4
          JP NZ,boucle  5
          RET           5
```

Pour comparer les deux boucles, nous pouvons ne pas tenir compte des instructions qui n'appartiennent pas à la boucle, car elles ne prennent pas un temps très significatif. La boucle du dernier programme prend 18,5 microsecondes alors que la première durait 22 microsecondes. C'est une amélioration incontestable mais pas encore extraordinaire.

Dans les routines que nous venons de décrire, la boucle est exécutée &4000 (16384) fois. A chaque exécution, le contenu de l'accumulateur, c'est-à-dire seulement 1 octet à chaque fois, est chargé dans une case mémoire de l'écran. Cela est effectué par l'instruction LD (DE),A. Malheureusement, il n'existe pas d'instruction dans le Z80 permettant de faire LD (DE),HL, ce qui nous permettrait de changer deux cases mémoires à la fois. Cela améliorerait considérablement le temps de traitement. Mais il existe des instructions permettant de sauvegarder des registres 16 bits. Elles ne permettent pas d'adresser aussi élégamment que le ferait l'instruction LD (DE),A, pour laquelle il suffirait d'incrémenter le registre DE.

Pour sauvegarder deux octets avec une seule instruction, il faut faire appel à un truc. Ce truc consiste à utiliser les fonctions de la pile. Normalement, la pile est utilisée pour sauvegarder momentanément les contenus de registres par exemple, lorsque l'on doit se brancher sur une autre routine qui utilise ces registres. La pile est donc située dans une zone mémoire protégée du BASIC. Le truc est simplement de mettre la pile dans le secteur de l'écran, pour pouvoir utiliser l'instruction de pile très rapide PUSH rr.

Cette instruction dure 6.5 microsecondes et elle est donc à peine plus rapide qu'un double LD (DE),A qui dure 7 microsecondes, mais elle modifie le compteur d'adresse en conséquence, ce qui n'est pas fait par l'instruction de chargement. Ainsi l'instruction PUSH doit être comparée à l'instruction LD (DE),A plus une double incrémentation qui dure en tout 13 microsecondes. Cet avantage se passe de commentaires.

L'instruction PUSH rr sauvegarde le contenu du registre symbolisé par rr. Ce registre peut être BC, DE, HL ou AF (et IX, IY). Le compteur de boucle a besoin seulement d'être de &200, car le nombre de boucles sera divisé par 2 avec PUSH rr. On sauvegardera &2000*2 octets au lieu de &4000*1 octets. Avant d'en venir au programme, il faut prévoir une cause d'erreur éventuelle. Le pointeur de pile (SP) est affiché à l'écran durant le déroulement du programme. Afin qu'il reprenne sa valeur première après le programme, il faut le mémoriser. On le sauvegardera donc dans deux cases mémoires qui ne seront pas modifiées. Nous utiliserons les cases mémoires &A030/31, qui se trouvent au-dessus de la mémoire adressable par BASIC. Après la boucle de notre programme, SP reprend sa valeur initiale afin de permettre un retour au BASIC sans problème. Si on oublie cela, le système se plantera inévitablement. Cela tient au fait que l'instruction CALL nn, que nous utilisons pour appeler notre routine, sauve l'adresse de retour ainsi que certains paramètres sur la pile, qui seront repris par l'instruction RET. Il est donc évident que SP doit reprendre son ancienne valeur après avoir été sauvegardé dans la mémoire écran.

Voici maintenant le programme :

```
         LD (&A030),SP  sauvegarde de SP
         LD SP,&FFFF    pile en mémoire écran
         LD HL,&3FFE    initialisation du compteur
         LD BC,&FFFF    valeur à charger
boucle   PUSH BC        modification de la mémoire
         DEC HL         décrémentation du compteur
         BIT 5,H        flag mis en place
         JP NZ,boucle   branchement conditionnel
         LD SP,(&a030)  reinitialisation de SP
         RET            retour au prg principal
```

La boucle de ce programme prend 18,5 microsecondes, comme la routine précédente. Mais il y a deux octets de sauvegardés à la fois, alors que dans la routine précédente, il n'y en avait qu'un. Cela montre l'efficacité de l'instruction PUSH. Certains vont peut être croire que l'instruction LD (&A030),SP n'adresse qu'une seule case mémoire alors que la valeur de SP est sur deux octets. Mais en fait cette instruction a été prévue pour cela. L'octet de poids faible est sauvegardé dans &A030 et l'octet de poids fort dans &A031. Cette forme de sauvegarde de valeur sur deux octets, un de poids fort et l'autre de poids faible est couramment utilisée. On sauvegarde toujours d'abord celui de poids faible (L) puis celui de poids fort (H). Il faut faire très attention à cela car si on l'ignore, beaucoup de programmes risquent de paraître incompréhensibles.

Avec l'instruction LD SP,(&a030), pour laquelle le transfert de données se fait à l'envers, c'est d'abord l'octet-L puis l'octet-H qui sont transférés.

Encore un exemple important pour cela: si il faut faire un branchement vers l'adresse &6533, il faut utiliser les codes CD 3365.

Pour finir ce chapitre, j'aimerais encore montrer une routine d'effacement très rapide. Avec le OU logique, le flag est mis à 1, ce qui conditionne le branchement.

```
        LD (&A020),SP   sauvegarde de SP
        LD SP,&FFFF     SP sur l'4cran
        LD BC,&FFFF     chargement de la valeur
        LD HL,&C001     dernière adresse
        XOR A           effacer l'accu
        LD (&C001),A    effacer la dernière adresse
boucle  PUSH BC         sauvegarde de la valeur
        OR (HL)         test si fini
        JP Z,boucle     branchement conditionnel
        LD SP,(&A020)   SP reprend ancienne valeur
        RET             retour au prg principal
```

D'abord, la valeur du pointeur de pile, qui est très importante, est sauvegardée dans le registre &A020. Cette mémoire est hors de la routine d'effacement.
Puis SP est mis dans l'écran (LD SP,&FFFF), le registre BC prend la valeur à sauvegarder (LD BC,&FFFF) et le registre HL prend la valeur de l'adresse de la dernière case mémoire (transformer (LD HL,&C001).
Puis l'accu prend la valeur 0 car on effectue un ou exclusif avec lui-même, ce qui est une méthode élégante qu'il faudra retenir. En effet, beaucoup de programmes en assembleur se servent de tels trucs, plus rapides mais plus difficiles à comprendre. Comme l'accumulateur est de nouveau nul, on peut l'utiliser pour effacer la case mémoire &C001, c'est-à-dire de lui donner la valeur 0. C'est important car la condition de branchement ne fonctionne que lorsque cette case mémoire et l'accumulateur sont nuls dès le départ.
Si vous avez des difficultés à comprendre, vous devriez simuler le fonctionnement de la boucle sur le papier, afin de saisir le rôle de l'instruction OR (HL). Ainsi la raison de cet effacement de la case mémoire et de l'accu devient claire. Dans la boucle, la condition de branchement n'est remplie que quand on effectue une comparaison entre l'accumulateur et une case mémoire. La case mémoire est adressée indirectement par le registre HL.

Le résultat de l'ordre OR est nul tant que l'accumulateur et la case mémoire &C001 sont nuls tous les deux. Dès que l'instruction PUSH met la valeur &FF dans &C001, le résultat de OR (HL) est différent de zéro et la condition n'est plus remplie, donc on quitte la boucle.

Parlons rapidement de la durée de cette boucle. Nous ne tiendrons pas compte des instructions hors de la boucle.

```
boucle   PUSH BC        6,5
         XOR  A         3,5
         JP   NZ,boucle   5
```

Une boucle dure 15 microsecondes, ce qui fait 0.246 secondes pour tout l'écran et 7,5 microsecondes par octet.

Avec le dernier programme, nous avons dû atteindre un maximum d'efficacité dans le remplissage de l'écran. Aucune autre structure ne devrait permettre d'aller plus rapidement avec une instruction de mémoire (PUSH BC) par boucle. Si nous augmentons le nombre d'instruction PUSH dans la boucle, le temps pris par les instructions JP NZ et OR A diminue. Avec 10 instructions PUSH par ligne, nous atteignons 3,7 microsecondes par octet, à comparer aux 7,5 microsecondes par octet avec un seul PUSH. Avec chaque PUSH en plus, nous nous approchons de la limite de 3,25 microsecondes qui est la moitié d'un PUSH. Mais avec une telle technique de programmation, nous limitons la mémoire utilisable. Il faut donc trouver un compromis.

Il reste seulement à expliquer pourquoi SP doit absolument être reinitialisé avant l'instruction RET. Le pointeur de pile doit prendre à nouveau sa valeur initiale, c'est-à-dire pointer sur la case mémoire dans laquelle étaient mémorisés les paramètres de retour. Un exemple excellent est l'instruction GOSUB-RETURN en BASIC. Si le BASIC rencontre une instruction GOSUB, il mémorise la ligne où elle se trouve afin de savoir où il doit retourner après un RETURN. Cette analogie n'est bien sûr pas hasardeuse. Pour des opérations de ce type, le CPC dispose d'une pile BASIC spéciale, qui fonctionne selon le principe de la pile LIFO.

Lors de la sauvegarde du contenu du registre double BC, nous ne sommes pas limités à la valeur &FF que j'ai utilisée pour notre exemple. Nous pouvons prendre n'importe quelle valeur de &01 à &FF, mais pas la nulle. Dans ce cas, cela conduirait à une boucle sans fin et nous aurions programmé un beau BUG!

Si vous vous sentez maintenant assez calé pour essayer de corriger cela, voici un petit conseil : la suite des instructions reste la même, il suffit de changer quelques valeurs. Ne vous découragez pas. Avant d'arriver à de grands succès, l'ordinateur se plantera bien des fois et donnera à votre programme apparemment parfait, l'allure d'une construction abhérante.

Si vous désirez insérer plusieurs instructions PUSH dans votre routine, notez qu'il ne faut pas pousser un octet directement avant l'adresse &C000. La conséquence en serait le plantage du système. Cela tient au fait que la zone mémoire située directement avant &C000, c'est-à-dire &BFFF contient la base de la pile. A la mise sous tension, la zone réservée à la pile est &BF00-&BFFF. Pour éviter de ne pas pousser un octet avant &C000, on peut donner à HL une valeur supérieure par exemple. Si vous ne comprenez pas, une simulation sur papier est toujours utile. Evidemment, il ne s'agit pas de faire le calcul pour tout l'écran, mais sur une partie, afin de comprendre le fonctionnement du programme. Il ne faudrait pas s'imaginer que ce type de test est un jeu d'enfant. Pour la recherche des erreurs et la mise au point des programmes, c'est très efficace.

Voici comme d'habitude le programme d'effacement le plus rapide avec un chargeur BASIC. A la place du branchement absolu, j'ai mis un branchement relatif, qui ralentit la routine d'une microseconde, mais qui la rend relogeable. 'relogeable' signifie qu'elle peut être inscrite n'importe où en mémoire.

```
5    mode 2:somme=0
10   for i=valeurdedepart to valeurdedepart+28
20   read valeur$:valeur=val('&'+valeur$)
30   poke i,valeur:somme=somme+valeur:next
50   if somme <> 3682 then print 'erreur dans les datas'
60   data ed,73,20,a0,31,ff,ff,01,ff,ff,21,01,c0,af
70   data 32,01,c0,c5,b6,28,fc,ed,7b,20,a0,c9,00,00
```

13.4. Les instructions puissantes du Z80

Le microprocesseur Z80 possède plusieurs centaines d'instructions. Cela peut paraître une limite insurmontable pour comprendre rapidement cette technique. Mais n'ayons pas de crainte. Une observation de plus près montre que l'on peut regrouper de nombreuses instructions, qui ont pratiquement la même fonction, mais qui possèdent des codes spécifiques en fonction des registres.

Avant de parler d'une instruction importante, je voudrais vous expliquer comment je montrerai l'infuence d'une instruction sur le registre des FLAGs. Sous l'abréviation des noms de FLAGs, se trouvent des caractères qui expliquent l'influence sur le FLAG.

```
S  Z H  P/V N   C
?  * 1  ?   0   - m/t/z
```

* : FLAG modifié en fonction du résultat
\- : Flag inchangé
1 : flag mis à 1
0 : flag effacé
? : flag modifié de façon imprévisible
m : nombre de cycles de machine
t : nombre de cycles d'horloge
z : temps nécessaire en microsecondes
b : numéro du bit
r : registre A, B, C, D, E, H, L

13.4.1 Instructions de manipulation des BITs

Nous allons maintenant expliquer environ 1/3 des instructions du Z80. Il s'agit des 168 instructions qui permettent de manipuler les BITs des registres A, B, C, D, E, H et L. Ils sont de la forme :

BIT b,r ; SET b,r et RES b,r

Nous connaissons déjà l'instruction BIT b,r, bien que nous ne l'ayons pas vue sous cette forme.

BIT b,r

Cette instruction teste le BIT (précisé par le paramètre) d'un registre.

Les paramètres b et r indiquent le numéro de Bit et le nom du Registre.

Le bit de poids fort d'un octet a toujours le numéro 7 et se trouve à gauche alors que le bit de poids faible aura le numéro 0 et se trouvera à droite. L'abréviation r indique qu'il s'agit d'un registre 8 bits dont nous avons déjà parlé.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	2/8/4

SET b,r

Cette instruction met le bit indiqué du registre voulu à l'état 1. Nous avons déjà expliqué les paramètres.

S	Z	H	P/V	N	C	
–	–	–	–	–	–	2/8/4

RES b,r

Cette instruction efface le bit, c'est à dire qu'elle le met à l'état 0.

S	Z	H	P/V	N	C	
–	–	–	–	–	–	2/8/4

Nous venons d'expliquer environ un tiers des instructions du Z80. Nous pouvons expliquer de la même façon un autre tiers des instructions. Ce qui restera prendra plus de temps et de place. Mais il n'est pas nécessaire de connaitre toutes les instructions dès le début. On devine rapidement devant un problème laquelle peut être utile. Il suffit alors de feuilleter l'index des instructions et de prendre celle qui apporte une solution simple. L'index dont nous parlons dépasse le cadre de cet ouvrage. Je convie les fans d'assembleur à se conférer aux ouvrages sur l'assembleur qui sont disponibles à un prix moyen. En ce qui concerne ce chapitre, il est inutile de faire une telle acquisition, car les instructions utilisées seront expliquées clairement.

Avant d'en venir à de nouvelles instructions, je vais expliquer quelques autres instructions de manipulation de BITs.

BIT b,(HL)   BIT b,(IX+d)   BIT b,(IY+d)

parlons rapidement de l'instruction :

BIT b,(HL)

Le registre HL contient dans ce cas l'adresse de la mémoire (non d'un registre du CPU, mais une case mémoire) dont le bit numéro b doit être testé. Ce mode d'adressage indirect permet de s'épargner des sauvegardes et des chargements longs.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	3/12/6

SET b,(HL) / RES b,(HL)

Ces instructions changent l'état du bit numéro b du registre HL et le mettent à 1 pour l'une et à 0 pour l'autre.

S	Z	H	P/V	N	C	
						4/15/7.5

13.4.2 Instruction de rotation et de deplacement

Toutes les instructions de rotation et de déplacement (sauf une) on les points suivants en commun:

a) Le contenu global des registres concernés est déplacé d'un bit à gauche ou à droite. Après l'instruction qui déplace le contenu à gauche, le contenu du BIT 0 se trouve dans le bit 1, celui du bit 1 dans le bit 2.

b) Le dernier bit à déplacer est mis dans le carry, ce qui n'est pas une solution provisoire mais qui a des raisons que nous développerons. Lors du déplacement vers la gauche, il s'agit du bit 7, pour le déplacement à droite, il s'agit du bit 0.

c) Logiquement, le flag de carry est modifié par ces instructions.

d) Toutes les instructions de déplacement et de rotation peuvent être utilisées avec les opérandes suivantes qui seront désignées par 's' :

A B C D E H L (HL) (IX+D) (IY+D)

RL s

Le contenu de l'opérande est déplacé vers la gauche. Le bit 7 est mis dans le carry et le contenu du carry est mis dans le bit 0.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RR s

Le contenu de l'opérande est déplacé vers la droite. Le bit 0 est mis dans le carry et le contenu du carry se retrouve au bit 7.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RLC s

Le contenu de l'opérande est déplacé vers la gauche. Le bit 7 se retrouve a la place du bit 0 et dans le carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RRC s

Le contenu de l'opérande est déplacé vers la droite. Le bit 0 se retrouve dans le bit 7 et dans le carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SLA s

Cette instruction déplace le contenu de l'opérande vers la gauche. Le bit 0 qui est alors libre prend la valeur 0. Le bit 7 est mis dans le carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRA s

Cette instruction déplace le contenu de l'opérande vars la droite. Le bit 0 est mis dans le carry. Le bit 7 reste inchangé.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRL s

Cette instruction déplace le contenu de l'opérande vers la droite. Le bit 0 est mis dans le carry. Le bit 7 est toujours effacé, c'est à dire mis à 0.

S	Z	H	P/V	N	C
*	*	0	*	0	*

Pour les instructions de déplacement et de rotation que nous venons de décrire, voici la table des temps en fonction de l'opérande s :

s :	m	t	z
r	2	8	4
(HL)	4	15	7,5
(IX+d)	6	23	11,5
(IY+d)	6	23	11,5

UTILISATIONS DES INSTRUCTIONS DE DEPLACEMENT ET DE ROTATION

Après avoir vu quelques uns (pas tous bien entendu) des ordres de déplacement et de rotation, venons en à des exemples pratiques d'utilisation de ces instructions.
Pour des raisons multiples, les transferts de données ne peuvent se faire que par un nombre réduit de cables. Naturellement, on ne peut faire passer qu'un seul bit par fil. Ce type de transfert de données bit par bit s'appelle le transfert série.
Nous n'allons pas étudier les particularités du transfert série mais plutôt voir comment diviser un octet en plusieurs bits afin de lui donner un format qui puisse permettre de le transférer par série. On peut l'expliquer simplement avec un exemple.

Il est précisé que l'octet à transférer se trouve dans le registre D.

```
        LD B,8          ;compteur 8 bits par octet
boucle  XOR A           ;effacement de l'accu
        SLA D           ;bit 7,d dans le carry
        JR NC,M1        ;saute l'instruction suivante
        LD A,1          ;accu différent de 0
m1      CALL affichage  ;branche vers la routine d'affichage
        DEC B           ;initialisation du compteur
        JP NZ,boucle    ;bit suivant
        RET
```

D'abord, le compteur B prend la valeur 8, afin de contrôler le nombre des bits transférés. Puis l'accumulateur est effacé par XOR A. Avec SLA D, le contenu du registre D est déplacé d'un bit vers la gauche ce qui met le bit 7 dans le carry. Si le bit 7 4tait nul, alors le bit de carry sera nul lui aussi. L'instruction JR NC,Ml est donc positive, alors on se branche sur ml. Ainsi l'accu demeure inchangé, donc nul. Si le bit 7 avait la valeur 1, le carry contiendrait un 1 et l'instruction JR NC,Ml serait négative donc pas exécutée. L'accumulateur prendrait donc la valeur 1 avec l'instruction suivante LD A,1.
Au label Ml, l'instruction CALL se branche sur une routine d'affichage, qui devrait être constituée de telle façon qu'elle envoie un 0 logique lorsque l'accu est nul et ua 1 logique lorsque l'accu est à 1. Il est inutile de montrer ici comment cela doit se passer et ce que sont des 1 et des 0 logiques.

Il est important de savoir comment on peut modifier le bit carry avec l'instruction SLA D pour pouvoir effectuer un branchement en fonction de son contenu et pour pouvoir fixer les paramètres de la routine de sortie en conséquence.

Un autre exemple d'utilisation de l'instruction SLA r est la multiplication par deux d'un registre ou du contenu d'une case mémoire.

Voici tout d'abord un exemple de ce qu'il ne faut pas faire pour programmer une telle routine: 13*2

```
          LD  B,13        opérande
          LD  C,2         multiplicateur
          XORA            effacer l'accu
boucle    ADDA,B          addition de l'opérande
          DECC            compter chaque passage
          JP  NZ,boucle   branche si C<>0
          RET
```

D'abord l'opérande et le multiplicateurs sont inscrits dans les registres respectifs B et C. Puis l'accumulateur est effacé afin de lui ajouter l'opérande. Dans la boucle 'boucle', on additionne c fois l'accumulateur avec la valeur du registre B.

Le listing qui suit conduit au même résultat :

```
LD A,13  accu=13
SLA A    rotation de A vers la gauche
RET
```

Expliquons rapidement ce type de multiplication avec le système binaire.

```
Registre A   00001101 =  8+4+1=13  (avant)
Registre A   00011010 = 16+8+2=26  (après)
```

A droite des nombres binaires sont inscrites les sommes des bits à 1 sous la forme décimale. Si on d4place les bits vers la gauche, chaque case de bit prend la valeur supérieure, c'est à dire le double. Lorsque l'on déplace les bits, la somme est donc multipliée par deux.

Question : Que se passe-t-il quand le nombre à multiplier par deux est plus grand que 127 ?

Dans ce cas, le carry a son importance. Mais pour pouvoir utiliser ce fait, il faut se servir d'un second registre qui récupérera les carrys déplacés. Prenons l'exemple 160*4, avec 160 dans le registre L. Ce registre doit être multiplié par deux deux fois de suite.

Mais il faut utiliser deux registres :

```
          LD H,0          effacer le registre H
          LD L,160        L=160
          LD B,2          compteur
boucle    SLA L           déplace L vers la gauche
          SLA H           déplace H vers la gauche
          OR A            effacement du carry
          DEC B           compter chaque passage
          JP NZ,boucle
          RET
```

D'abord H est effacé, L prend la valeur 160 et B est utilisé comme compteur. Dans la boucle qui suit, le registre L est toujours déplacé et son contenu change l'état du carry en se déplaçant. Le carry est alors poussé dans H. Etant donné qu'on peut utiliser les registres H et L comme un registre 16 bits, on a le résultat sur 16 bits dans HL.
L'instruction OR A est présente pour rendre le programme plus lisible. Elle efface un carry éventuel de H car celui ci serait poussé dans le bit 0 du registre L.

Pour éclaircir le fonctionnement de cette forme spéciale de calcul, nous pouvons présenter cette opération sous forme binaire.

```
registre    H           L
            00000000    10100000 = 128+32=160 (avant)
            00000001    01000000
            00000010    10000000 = 512+128=640 (aprés)
```

Si vous avez déjà vu la liste d'instructions du Z80, vous devez penser qu'il n'existe pas d'instruction de déplacement ou de rotation sur 16 bits.

Mais en fait il en existe une.

Dans l'exemple que nous venons de décrire, nous avons substitué à une instruction de multiplication inexistante, un système de déplacement de registre vers la gauche. Revoyons le problème et créons une instruction de déplacement avec une instruction d'addition.

Comme nous le savons, on peut considérer le registre double HL comme un quasi accumulateur 16 bits. On peut additionner les registres BC, DE, SP et HL au registre HL. Comme on peut additionner HL à lui même, ce qui correspond à la multiplication par deux, le résultat équivaudra à un déplacement vers la gauche.

L'instruction 16 bits ADD HL,HL correspond donc à l'instruction sur 8 bits SLA r. Le contenu des registres considérés est déplacé vers la gauche et le bit de poids fort est mis dans le carry alors que le bit de poids faible prend la valeur 0.

13.4.3 Instruction d'Arithmétique sur 16 BITS

ADD HL,rr

Le contenu du registre double désigné par rr est additionné au contenu de HL et mis dans le registre HL. rr peut désigner les registres BC, DE, HL et SP. Le bit half carry est fixé par un déplacement du bit 11 vers le bit 12, c'est-à-dire du nibble de poids faible du registre H vers son nibble de poids fort. Un déplacement de L vers H n'est pas indiqué.

S	Z	H	P/V	N	C	
-	-	*	/	0	*	3/11/5.5

Avec cette instruction, on peut effectuer une rotation sur 16 bits, ce qui n'est pas implémenté dans le Z80.

Comme avec le véritable accumulateur, on peut effectuer une addition par le carry avec le registre double HL, outre l'addition ordinaire.

L'instruction pour cela est :

ADC HL,rr

Le contenu du registre HL et celui du registre spécifié par rr sont ajoutés l'un à l'autre. On additionne en outre le carry. rr peut être BC, DE, HL et SP. Tous les flags, excepté le flag N sont modifiés en fonction du résultat.

S	Z	H	P/V	N	C	
*	*	*	*	0	*	4/15/7.5

Avec l'instruction sur 16 bit qui suit, nous en aurons fini avec les possibilités arithmétiques du registre HL.

SBC HL,rr

Le contenu du registre désigné par rr est soustrait du contenu du registre HL. Le résultat de cette soustraction est chargé de nouveau dans HL. rr peut être BC, DE, HL ou SP. Excepté le flag N, tous les flags sont modifiés en fonction du résultat.

S	Z	H	P/V	N	C	
*	*	*	*	1	*	4/15/7.5

13.4.4 Instructions de blocs

Lorsque l'on veut transférer un secteur de la RAM à une autre place, il est possible d'utiliser des instructions du Z80. Il s'agit de:

LDI	chargement avec incrémentation
LDIR	chargement avec incrémentation et répétition
LDD	chargement avec décrémentation
LDDR	chargement avec décrémentation et répétition

Chacune de ces instructions est constituée en principe de plusieurs instructions assembleur qui sont exécutées à la suite.

INSTRUCTIONS DE TRANSFERT DE BLOCS

LDI

Le contenu d'une case mémoire précisée par HL est chargé dans une autre case mémoire précisée par DE. Puis les registres double HL et DE sont incrémentés. Enfin BC est décrémenté.

S	Z	H	P/V	N	C	
-	-	0	*	0	-	4/16/8

LDIR

LDI LD(DE),(HL); INC HL; INC DE; DEC BC jusqu'à BC=0

Le contenu de la case mémoire précisée par HL est chargé dans la case mémoire adressée par DE. Puis les registres doubles HL et DE sont incrémentés. Enfin, BC est décrémenté.
Cette suite d'instructions est exécutée jusqu'à ce que le registre BC soit nul.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0)
						5/21/10.5 (BC<>0)

LDD

LDI LD(DE),(HL); DEC HL; DEC DE; DEC BC

Le contenu de la case mémoire précisée par HL est chargé dans la case mémoire adressée par DE. Puis les registres doubles HL et DE sont décrémentés. Enfin, BC est décrémenté.

S	Z	H	P/V	N	C	
-	-	0	*	0	-	4/16/8

LDDR

LDI LD(DE),(HL); DEC HL; DEC DE; DEC BC jusqu'à BC=0

Le contenu de la case mémoire qui est indiquée par HL est chargé dans la case mémoire adressée par DE. Puis les registres doubles HL et DE sont décrémentés. Enfin, BC est décrémenté. Cette séquence d'instructions est exécutée jusqu'à ce que le registre BC soit nul.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0)
						5/21/10.5 (BC<>0)

Voici un résumé:

```
LDI   LD(DE),(HL);  INC HL;  INC DE;  DECBC
LDIR  LD(DE),(HL);  INC HL;  INC DE;  DECBCjusqu'àBC=0
LDD   LD(DE),(HL);  DEC HL;  DEC DE;  DECBC
LDDR  LD(DE),(HL);  DEC HL;  DEC DE;  DECBCjusqu'àBC=0
```

Pour tous les possesseurs d'un CPC 464 ou 664 qui envient les instructions de transfert de mémoire du 6128, voici une routine qui permet de copier des blocs de mémoire.

```
LD BC,&4000   initialisation du compteur
LD DE,&C000   adresse objet
LD HL,&4000   adresse source
LDIR          instruction de chargement de bloc
RET
```

Vous pouvez utiliser ce programme très utile avec le chargeur BASIC qui suit.

```
 5 mode 2:somme=0
10 for i=valeurdedepart to valeurdedepart+11
20 read valeur$:valeur=val('&'+valeur$)
30 poke i,valeur:somme=somme+valeur:next
40 if somme<>985 then print 'erreur dans les datas'
50 data 01,00,40,11,00,40,21,00,c0,ed,b0,c9
```

— —

Après avoir chargé la routine avec ce programme, vous pouvez l'utiliser avec un CALL.

Ormis le READY, rien ne semble changé. Mais en fait, cette routine a copié le bloc de mémoire n0 3, c'est-à-dire la mémoire d'écran, dans le bloc 1 de la RAM.

Echangez maintenant les valeurs soulignées dans la ligne de datas. Il s'agit des octets de poids fort de l'adresse de départ des blocs. Après avoir remis le programme en route, le bloc 1 sera copié dans la mémoire d'écran avec CALL. Avant de lancer le programme par CALL, je vous conseille d'effacer l'écran par mode 2, afin d'avoir un effet plus évident.

Vous avez le choix de la zone mémoire où vous voulez copier la mémoire d'écran.

Il faut seulement faire attention à ne pas écraser les programmes BASIC ni la mémoire réservée pour le système d'exploitation. La longueur du bloc reste libre. Dans ce cas, j'ai choisi la taille de la mémoire écran.

XIV. QUELQUES ROUTINES EN LANGAGE MACHINE POUR MANIPULER L'ECRAN

14.1 Scrolling de l'écran en douceur.

Lorsque l'ordinateur doit afficher une nouvelle ligne à l'4cran tout en bas, il fait de la place afin de pouvoir déplacer les lignes vers le haut. La hauteur d'une ligne de caractères fait 8 pixels. L'ordinateur ne déplace pas pour autant le contenu de la mémoire, mais il change l'affichage. La mémoire dont le contenu est affiché au coin gauche en haut n'est alors plus &C000. L'ordinateur utilise donc une adresse de base de l'écran. Avec le petit chargeur BASIC que voici, vous pouvez tester ce type de scrolling :

```
10  for i%=&A000 to &a005
20  read valeur$:valeur=val('&'+valeur$)
30  poke i%,valeur
30  next
40  data 06,01,cd,4d,bc,c9
            —
```

Si l'écran doit se dérouler vers le bas, il suffit de mettre la valeur soulignée à 0. Vous devrez toujours faire attention à ce que l'écran n'ait pas encore été déroulé, car cela conduirait à des résultats incontrôlables. Voici maintenant un chargeur BASIC qui réalise un scrolling qui déplace chaque ligne de pixels une à une. Le contenu est déplacé vers le haut.

```
10  mode 2:somme=0
20  for i%=&A000 to &A036
30  read valeur$:valeur=val('&'+valeur$)
40  poke i%,valeur:somme=somme+valeur
50  next
60  if somme<>5695 then print'erreur dans les datas'
100 data 01,c7,c7,21,00,c0,22,00,b0,c5,2a,00,b0,54,5d,cd
110 data 26,bc,22,00,b0,01,50,00,ed,b0,c1,cd,09,bb,38,15
120 data 0d,20,e6,c5,11,80,ff,21,36,a0,01,50,00,ed,b0,c1
130 data 0e,c7,05,20,ce,c9,00,00,00
```

Si vous voulez déplacer l'écran vers le bas, remplacez les lignes 100 à 110 comme suit :

```
100 data 01,c8,c7,21,80,ff,22,00,b0,c5,2a,00,b0,54,5d,cd
110 data 29,bc,22,00,b0,01,50,00,ed,b0,c1,cd,09,bb,38,15
120 data 0d,20,e6,c5,11,00,c0,21,36,a0,01,50,00,ed,b0,c1
```

La somme doit être dans ce cas de 5699.

Si maintenant vous ne voulez pas déplacer tout l'écran mais seulement une partie à gauche de l'écran, exécutez les instructions suivantes avant le programme qui précède:

```
poke &a016,10 : poke &a02b,10
```

Le 10 correspond à la largeur du secteur à déplacer.

Voici enfin le programme sous une forme plus courte et plus rapide. Il est beaucoup plus rapide mais ne peut pas être arrêté. Il s'arrête de lui-même après le scrolling de tout l'écran. Le programme chargeur utilisé précédemment peut servir de nouveau. La boucle FOR-NEXT doit avoir pour limites &A000 et &A02F et la somme est 4368. Les lignes de datas doivent être adaptées ainsi :

```
100 data 01,c8,c8,21,80,ff,22,00,b0,c5,24,00,b0,54,5d,01
110 data 00,08,ed,42,cb,74,20,04,01,b0,3f,09,22,00,b0,01
120 data 50,00,ed,b0,c1,0d,20,e1,0e,c8,05,20,d6,c9,00,00
```

14.2 Srolling latéral de la dernière ligne d'écran

Pour la présentation de textes, plusieurs effets sont possibles. On peut inverser ou colorer les caractères, les encadrer, les souligner, les faire clignoter, etc... Un effet intéressant et nouveau est celui de laisser se dérouler une ligne qui affiche le texte. Avec le programme qui suit, vous pourrez faire un scrolling de la ligne inférieure de l'écran, qui se déroulera de la droite vers la gauche.

```
A000  117702 ANF  LD   DE,&277       une exécution
A003  D5          PUSHDE
A004  114f08      LD   DE,&84f  adresse de début de l'opérande
A007  ED5341A0    LD   (&A041),DE   sauvegarde de l'op.
A00B  01084F M2   LD   BC,&4F08     compteur 8 Z/ mem &4F
A00E  21CEC7      LD   HL,&C7CE     adresse de début
A011  B7          OR A            efface carry
A012  CB16   M1   rl   (HL)
A014  2B          DEC HL          nouvelle adresse
A015  05          DEC B           ligne finie ?
A016  20FA        JR   NZ,M1
A018  114F00      LD   DE,&004F
A01B  3005        JR   NC,M3      fin de ligne dans carry ?
A01D  19          ADD HL,DE       ancienne adresse de début
A01E  CBC6        SET  0,(HL)     bit de droite positionné
A020  1803        JR   M4         saut après RES
A022  19     M3   ADD HL,DE       ancienne adresse de début
A023  CB86        RES  0,(HL)     effacer bit droite
A025  ED52   M4   SBC  HL,DE
A027  064F        LD   B,&4F      compteur de ligne init.
A029  ED5B41A0    LD   DE,(&A041)  prendre opérande
A02D  19          ADD HL,DE
A02E  0D          DEC  C
A02F  20E1        JR   NZ,M1
A031  D1          POP  DE         compteur du nbre de mem
A032  1B          DEC  DE
A033  CD09BB      CALL&BB09       touche pressée?
A036  3808        JR   C,fin      arrêt
A038  D5          PUSH DE
A039  CB7A        BIT  7,D        test fin de compteur
A03B  28CE        JR   Z,M2
A03D  D1          POP  DE         correction de pile
A03E  18C0        JR ANF          nouveau tour
A040  C9          fin RET
A041              mem d'opérande
A042              mem d'opérande
```

L'écran ne doit pas avoir subi de scrolling avant l'appel de cette routine car l'affichage serait transformé. Le programme a été prévu pour le mode 2. Les autres modes donneront des résultats inutiles mais intéressants. Avec l'instruction de branchement relatif, ce programme est relogeable. Il peut être placé partout sur la RAM libre. Mais il faut faire attention aux mémoires &A031 et &A042. Le registre DE y est mémorisé temporairement. Si le programme est chargé sur ces mémoires, il faudra utiliser d'autres cases mémoire.

Le chargeur BASIC :

```
10  mode 2:somme=0
20  for i=&A000 to &a042
30  read valeur$: valeur=val('&'+valeur$)
40  poke i,valeur:somme=somme+valeur
50  next i
60  if somme<>6591 then print 'erreur dans les datas'
70  end
100 data 11,77,02,d5,11,4f,08,ed,53,41,a0,01,08,4f,21
110 data ce,c7,b7,cb,16,2b,05,20,fa,11,4f,00,30,05,19
120 data cb,c6,18,03,19,cb,86,ed,52,06,4f,ed,5b,41,a0
130 data 19,0d,20,e1,d1,1b,cd,09,bb,38,08,d5,cb,7a,28
140 data ce,d1,18,c0,c9,00,00,00
```

Après avoir exécuté le programme, la routine est en mémoire et peut être démarrée par CALL &A000 et être arrêtée en frappant sur une touche. Le programme chargeur ne sera plus utilisé et peut être effacé par DELETE 1-. Si vous avez protégé votre programme en langage machine par MEMORY &A000-1, le programme chargeur peut être effacé par NEW.

Un exemple d'utilisation de la routine :

```
10  locate 5,25: print 'recommencez la saisie'
20  CALL &A000
30  locate 1,25: print string$(80,' ');
40  mode 2
50  input 'nouvelle saisie';a
60  rem ....etc...
```

14.3. Copie d'écran pour 464/664

Cette section est spéciale pour les possesseurs d'un 464 ou d'un 664 et qui n'ont pas les possibilités de gestion d'écran du 6128. Avec SCREENCOPY, vous pourrez copier le contenu d'un bloc de RAM dans un autre bloc. Un bloc est un ensemble de cases mémoire qui fait 16koctets. Etant donné que la RAM fait 64k, elle est composée de quatre blocs.

	Adresse de début	de fin
bloc 0	&0000	&3fff
bloc 1	&4000	&4fff
bloc 2	&8000	&bfff
bloc 3	&C000	&ffff

Bien que l'utilisateur dispose de 42000 octets à l'allumage de l'ordinateur, il ne peut utiliser que le bloc 1 en entier.

Remarques sur les blocs :

Dans le bloc 0 se trouvent des routines indispensables du système d'exploitation. Le reste est utilisable librement. Pourquoi indispensables ? Essayez de mettre la valeur 0 dans la case 8 : poke 8,0... Mais attention de ne pas avoir un programme ou des données importantes dans l'ordinateur!

Comme nous l'avons déjà indiqué, le bloc 1 est entièrement disponible.

Dans le bloc 2, les mémoires de &8000 à &A67B peuvent être utilisées. Le secteur &a67C-&BFFF (42620-49151) est utilisé par le système d'exploitation. Les transformations dans ce secteur doivent être faites avec beaucoup de prudence, c'est-à-dire qu'aucune donnée importante ne doit se trouver dans l'ordinateur, et il faut éteindre le lecteur de disquettes. Cela peut conduire au plantage du système.

Le bloc 3 contient les informations concernant l'écran. Si on change son contenu, l'affichage est changé.

Voici un exemple pour montrer comment agit l'instruction SCREENCOPY: on peut utiliser le programme qui a été montré à la fin de la description de l'instruction LDIR. Dans ce programme, le secteur d'écran était copié dans le bloc 1. Si on avait sauvegardé auparavant une image dans ce bloc, il suffit de quelques millisecondes pour changer l'affichage.

Explication :

Après la mise en route de l'ordinateur ou après un reset, les informations concernant le moniteur se trouvent dans le bloc 3, donc dans la mémoire d'écran qui commence à l'adresse &C000. Lorsque l'on change l'adresse de début, par exemple en &4000 (adresse du bloc 1), l'image est faite avec les informations situées dans les 16383 cases suivant &4000. Si il s'y trouve une autre image, celle-ci est affichée. Avec le programme qui suit, vous pourrez changer l'adresse de base de l'écran.

```
10 for i=&A000 to &A005
20 read valeur$
30 valeur=val('&'+valeur$):poke i,valeur
40 next
50 data 3e,00,cd,08,bc,c9
60 input 'quel bloc 1/3),b
70 if b=1 then poke &A001,&40 else poke &A001,&C0
80 CALL &A000
```

Le listing désassemblé de la routine est:

```
A000  3E xx      LD A,xx
A002  CE 08 BC   CALL BC08
A005  C9         RET
```

Dans la première ligne, l'accu prend la valeur de l'octet significatif de l'adresse de l'écran. L'accu contient donc les paramètres pour la routine SCR SET BASE, qui sera appelée par CALL &BC08. Le RET se passe de commentaires.

Une variante intéressante pour modifier le contenu de l'écran peut être obtenue en arrêtant le programme en appuyant deux fois sur ESC puis en exécutant mode2 et CALL &BC08.
Sur l'écran apparaissent plusieurs lignes de points. La rangée supérieure représente les routines indispensables du système et les quatre suivantes, le programme BASIC.
Vous avez donc mis le secteur écran dans le premier bloc (bloc 0) et vous voyez les parties du programme qui s'y trouvent.
Entrez donc quelques lignes BASIC et voyez ce qui se passe lorsque vous faites ENTER.

14.4. SCREENSWAP pour le 464/664

La différence entre SCREENCOPY et SCREENSWAP tient au fait qu'avec SWAP le contenu des deux secteurs est échangé. Lors du COPY, le contenu d'un secteur est copié dans un autre, ce qui fait que les deux sont alors identiques.
Nous savons que lors de l'échange de deux variables, nous avons besoin d'une troisième.

Exemple : échange du contenu de A et B (x= variable transitoire)

x=A:A=B:B=x

Si nous voulons maintenant échanger deux blocs, nous ne possédons pas de troisième bloc. Par conséquent, nous devrons faire l'échange par petits morceaux. Ces morceaux font &100 octets de long dans notre exemple. Ils pourraient avoir une autre longueur mais ne pas être trop courts car la routine serait trop lente. Si les morceaux sont trop longs, il risquent d'occuper trop de place mémoire (qui est déjà réduite d'1/4 par le nouvel écran).

Le chargeur BASIC de SCREENSWAP 464/664 :

```
5   start=&A000:somme=0
10  for I%=start to start+&40
20  read valeur$:valeur=val('&'+valeur$)
30  poke i%,valeur:somme=somme+valeur:next
40  if somme<>5098 then print 'erreur dans les datas'
50  data 21,00,c0,22,00,81,21,00,40,22,02,81,01
60  data 40,00,c5,01,00,01,11,00,80,2a,00,81,ed
70  data b0,01,00,01,ed,5b,00,81,2a,02,81,ed,b0
80  data 01,00,01,ed,5b,02,81,21,00,80,ed,b0,21
90  data 01,81,34,23,23,34,c1,0d,c5,20,d1,c1,c9
```

Le listing assembleur de SCREENSWAP 464/664 :

```
A000    21 00 c0          LD    HL,&C000
A003    22 00 81          LD    (&8100),HL
A006    21 00 40          LD    HL,&4000
A009    22 02 81          LD    (&8102),HL
A00C    01 40 00          LD    BC,&0040
A00F    C5                PUSH BC
A010    01 00 01  boucle  LD    BC,&0100
A013    11 00 80          LD    DE,&8000
A016    2A 00 81          LD    HL,(&8100)
A019    ED B0             LDIR
A01B    01 00 01          LD    BC,&0100
A01E    ED 5B 00 81       LD    DE,(&8100)
A022    2A 02 81          LD    HL,(&8102)
A025    ED B0             LDIR
A027    01 00 01          LD    BC,&0100
A02A    ED 5B 02 81       LD    DE,(&8102)
A02E    21 00 80          LD    HL,&8000
A031    ED B0             LDIR
A033    21 01 81          LD    HL,&8101
A036    34                INC   (HL)
A037    23                INC   HL
A038    23                INC   HL
A039    34                INC   (HL)
```

```
A03A C1          POP BC
A03B 0D          DEC  C
A03C C5          PUSH  BC
A03D 20 D1       JR  NZ,boucle
A03F C1          POP  BC
a040  C9         RET
```

Le programme lui même est relogeable. Mais il ne peut pas être mis n'importe où. D'abord, il ne faut pas le mettre dans le bloc 1, car ce bloc sert à la sauvegarde de l'écran. De plus, la mémoire transitoire pour le SWAP et quatre cases mémoires réservées prennent les secteurs &8000-&8103. Ainsi, il ne faut pas utiliser les secteurs &4000-&8103 pour le programme.

Etant donné que vous avez fixé l'adresse de départ dans le chargeur BASIC à &8104, vous avez créé un secteur de mémoire dans lequel toute la place est occupée pour le SWAP (&4000-&8144).

XV. UN REMPLACEMENT SIMPLE DU BASIC PAR LES REGISTRES DU Z80

Il est bien connu que les routines du CPC sont parfois plus utiles lors de la programmation que leurs équivalents en BASIC.
Un bon exemple en est la routine qui est appelée avec CALL &bb06. Cette routine attend qu'une touche soit pressée. Puis le programme revient au BASIC. La routine &BB06 n'a pas besoin de paramètres.

Avec l'instruction SPEED WRITE x, on peut choisir la vitesse d'écriture sur la cassette, avec pour paramètres 1 (plus rapide) et 0. Avec 1, la sortie se fait en 2000 bauds. C'est le maximum autorisé par le CPC.
Des vitesses de transfert supérieures peuvent être atteintes avec la routine désignée par &BC68. Pour cela il faut fixer 3 registres du Z80. Avec le programme qui suit, les registres sont chargés facilement.

```
10  mode 2:somme=0:gosub 100
20  for i=1 to 10
30  read reg$,place$: print reg$;:input' ';valeur$
40  if valeur$' then valeur$='00'
50  valeur=val('&'+valeur$):place=val('&'+place$)
60  poke &A000+place,valeur
70  next
80  CALL &A00:end
100 for i=&A000 to &A012:read W$:W=val('&'+w$)
110 poke i,w:somme=somme+w:next
120 if somme<>960 then print 'erreur dans les datas':end
130 return
140 data 21,0,0,e5,f1,1,0,0,11,0,0,21,0,0,cd,0,0,c9,0
150 data a,2,f,1,b,7,c,6,d,a,e,9,h,d,l,c,hb,10,lb,f
```

La boucle des lignes 100 et 110 et les datas de la ligne 140 créent un petit programme à partir de la case mémoire &A000. Le programme demande les valeurs qui doivent être chargées dans les registres A, F, B, C, D, E, H et L. Les données suivantes HB et LB indiquent l'octet de poids fort et l'octet de poids faible.
Les saisies qui ne vous intéressent pas peuvent être passées avec ENTER.

Avec cet outil simple, nous avons une possibilité de connaître et d'utiliser le système du CPC. Revenons à la vitesse d'écriture du lecteur de cassettes. Les paramètres sont un octet à moitié nul pour HL et le test de longueur pour A.

1000 bauds :	HL=333	A=25
2000 bauds :	HL=167	A=50

Ces valeurs donnent une sécurité de transfert optimale. La rapidité maximale de transfert dépend de la qualité de la cassette.

XVI. LE CHARGEMENT DE PROGRAMMES EN LANGAGE MACHINE

Vous connaissez sans doute la méthode courante de chargement des programmes en langage machine. Les lignes de data sont écrites au dessus de la RAM BASIC avec poke. Auparavant, la mémoire nécessaire aura été réservée avec MEMORY. Cette méthode à un inconvénient :

Etant donné que la définition de caractères est sauvegardée au-dessus du BASIC, cela peut entraîner des recouvrements. Etant conscient de ce problème, les programmeurs du système ont construits une barrière.
Si le haut de la mémoire a été réservé par MEMORY, alors chaque SYMBOL AFTER entraîne un message d'erreur 'improper argument'. On évite ainsi que des programmes en langage machine ne soient effacés par une redéfinition des caractères.
D'un côté, c'est utile pour la protection des programmes en langage machine, mais d'un autre côté, on ne peut plus utiliser SYMBOL AFTER. Cela signifie que tout SYMBOL AFTER entraînera l'arrêt du programme en fonctionnement.
Une solution est de faire SYMBOL AFTER 0 dès le départ en mode direct, mais pour des programmes 'honnêtes', ce n'est pas une solution.

Il existe une seconde solution qui est d'écrire l'instruction en première ligne puis de l'effacer.

Pour le CPC 464, cela donne :

```
10 symbol after 100
20 a=peek(&ae81)+256*peek(&ae82)+1
30 poke a+4,&c5
40 for i=a+5 to a+peek(a)+256*peek(a+1)-2:poke i,32:next
50 memory &a000
.
.
.
```

pour le CPC 664/6128 :

```
10 symbol after 100
20 a=peek(&ae64)+256*peek(&ae65)+1
30 poke a+4,&c5
40 for i=a+5 to a+peek(a)+256*peek(a+1)-2:poke i,32:next
50 memory &a000
.
.
.
```

Une autre méthode serait de changer le pointeur HIMEM afin d'autoriser les symbol after même au milieu du programme. Si vous utilisez cette méthode, vous devrez faire attention à ce que votre programme en langage machine ne soit pas écrasé par la redéfinition de caractères.

Pour le CPC 664 et CPC 6128 :

```
10 memory &9fff
20 rem...
30 oldhimem=peek(&ae5e)+256*peek(&ae5f)
40 poke &ae5e,peek(ae60):poke &ae5f,peek(&ae61)
50 poke &b735,0
60 symbol after 200
70 memory oldhimem
80 rem....
```

Pour CPC 464 :

```
10 memory &9fff
20 rem...
30 oldhimem=peek(&ae7b)+256*peek(&ae7c)
40 poke &ae7b,peek(&ae7d):poke &ae7c,peek(&ae7e)
50 poke &b295,0
60 symbol after 200
70 memory oldhimem
80 rem...
```

Etant donné que toutes ces méthodes ne sont que des instructions, vous apprenez uniquement comment les utiliser de manière un peu originale. Maintenant nous allons vous montrer deux possibilités qui fonctionnent selon le même principe mais qui utilisent d'autres places mémoire.

Au lieu de charger le programme en langage machine au-dessus du BASIC, il est possible de programmer au-dessous du BASIC. Le début du pointeur de basic moins un est à l'adresse &ae81/2 (pour 664/6128: &ae64/5). Normalement, il pointe sur l'adresse &16f, c'est-à-dire que les programmes BASIC commencent à l'adresse &170.

Donnons une plus grande valeur à ce pointeur que nous appelerons lowmem. Alors, le secteur entre &170 et lowmem peut être utilisé pour des programmes en langage machine.

```
poke &ae64,int(lowmem/256)
poke &ae65,lowmem-int(lowmem/256)*256
new
```

(464 : mettre &ae81/2 au lieu de &ae69/5)

New doit être entré immédiatement aprés, car autrement divers pointeurs (par exemple pour le tableau de variables) pourraient être mal fixés. La libération du secteur n'est pas sensible en BASIC. Même symbol after fonctionne sans problèmes. Mais la place située sous le BASIC est libre pour des programmes en langage machine.

Il est possible, dans certaines limites, de sauvegarder des programmes en langage machine dans la RAM système. Un tel secteur de ram est la mémoire de programmation des touches. Son adresse de début se trouve en &b4el/2 (664/6128: &b62b/c). Normalement, le tableau commence à l'adresse &b446 (664/6128: &b590); elle fait 152 octets. L'inconvénient est que l'utilisation de l'instruction KEY n'est plus possible. Afin d'éviter cela, il faut redonner leur fonction normale aux touches avec KEY DEF.

Il est possible également de mémoriser le tableau des caractères définis par l'utilisateur. Cependant, les définitions de caractères effacées par l'utilisateur ne seront plus disponibles. Le code du premier caractère du tableau de symboles est à l'adresse &b294 (664/6128 : &b734). La taille est donnée par :

```
print (256-peek(&b294))*8 pour le 464
print (256-peek(&b734))*8 pour 664/6128
```

Evidemment, les codes 32 à 127 doivent demeurer inchangés une fois définis.

L'adresse de début du tableau est à l'adresse &B296/7 (664/6128 &b736/7), l'adresse de fin est située à l'adresse &b096/7 (664/6128 : &b075/6). Enfin, il est possible également de mettre un programme en langage machine dans la ram vidéo (adresses &C000 à &FFFF). Avec les indications suivantes :

La ram vidéo fait 16K (16K=16*1024=16384). En mode 2, il y a seulement 25*80*8 points d'utilisés. La différence (8*48=384 octets) est inutilisée. Lors de la mise en marche, avec l'instruction mode, les octets inutilisés sont situées tous les 48 octets dans les secteurs &c7d0-&c7ff, &cfd0-&cfff, &d7d0-&d7ff,etc... Le problème est qu'il ne faut pas exécuter de scrolling de l'écran.

Autrement, les secteurs libres que nous venons d'indiquer seraient déplacés et le programme serait écrasé. Malgré tout, cela peut être une bonne place pour la sauvegarde d'un programme car, exceptée celle que nous avons indiquée, il n'y a pas de restriction.

Nous allons maintenant donner deux autres méthodes qui ne fonctionnent qu'avec certains programmes en langage machine. Tous les programmes en langage machine qui doivent être sauvegardés doivent être relogeables, c'est-à-dire qu'ils ne doivent utiliser aucune adresse absolue qui pointe sur le programme lui-même. Souvent il est possible de ne pas utiliser d'adressage absolu, en n'utilisant que des branchements relatifs par exemple.

Les programmes doivent avoir cette particularité car nous les laissons 'dans les mains du système' qui les gère. On garantit seulement dans ce cas qu'aucun octet du programme ne sera modifié. Mais l'adresse de début est variable et sera modifiée en fonction des besoins.

Comment cela est-il possible ?

C'est très simple : le BASIC permet de gérer des listes d'octets, en l'occurence, des chaînes de caractères. Une chaîne de caractères est constituée d'une suite de codes. Sa longueur peut atteindre 255 caractères. Par conséquent, nous pouvons sauvegarder un programme machine en tant que chaîne de caractères.

Avec une boucle, nous allons lire ces codes dans des lignes de datas et les additionner à une chaîne avec des CHR$. p. ex. :

```
MAPRO$=MAPRO$+CHR$(octet)
```

Ainsi le programme en langage machine est mémorisé. L'appel est fait avec la fonction @, qui pointe sur le descripteur de chaîne. Pour notre programme, qui est dans MAPRO$, on peut écrire :

```
CALL PEEK(@MAPRO$+1)+256*PEEK(@MAPRO$+2)
```

La deuxième solution, construite sur le même principe, permet la sauvegarde de programmes de n'importe quelle longueur.

Cette fois, nous utilisons une variable du type integer. Un chiffre integer est constitué d'octets de poids fort et de poids faible. Les valeurs correspondant aux indices sont mémorisées dans un ensemble. Pour cette raison, nous pouvons utiliser des ensembles de chiffre INT pour mémoriser nos programmes en langage machine. Vous en saurez plus sur les variables de champ et leur traitement interne au chapitre 22. Le programme qui suit éclaircit le principe.

Il faut noter que :

1) Le champ doit être du type integer (%).

2) Si le programme a une taille variable, il faut rajouter des &00 aux lignes de datas.

```
10  longueur=26: ' nombre d'octets dans le programme
20  DIM variable%(INT((longueur-1)/2))
30  FOR i=0 to INT ((longueur-1)/2)
40  READ l$,h$
50  variable%(i)=VAL('&'+h$+l$)
60  NEXT i
70  END
80  DATA 3e,01,cd,0e,bc,cd,5,b9
90  DATA 7c,21,57,86,fe,01,28,06
100 DATA 2e,5c,38,02,2e,77,cd,0b
110 DATA 00,c9
120 REM appel avec CALL@variable%(0)
```

Après RUN, le programme peut être mis en route avec CALL @variable%(0). Lors de l'appel, l'indice donné doit être 0.

Avez-vous testé le programme ? Peut être avez-vous eu peur et croyez-vous que l'ordinateur s'est planté et a exécuté un RESET.
Ce n'est pas le cas. Seul le message de mise en route a été affiché. Le petit programme est très intéressant car il reconnaît automatiquement la version de l'ordinateur. Cela est rendu possible par la routine système 'KL version number'.

Voici le listing assembleur du programme :

```
A000          10  ; affichage
A000  3E01    20  LD A,1
A002  CD0EBC  30  CALL &BC0E  ; scr mode a
A005  CD15B9  40  CALL &B915  ; KL version number
A008  7C      50  LD A,H      ; BASIC version 0,1 ou 2
A009  215786  60  LD HL,%8657 ; pour 664
A00C  FE01    70  CP 1
A00E  28FE    80  JR z,ok
A010  2E5C    90  LD L,&5C    ; pour 464
A012  38FE   100  JR  C,ok
A014  2E77   110  LD  L,&77   ; pour 6128
****  ligne  80: OK=&A016
****  ligne 100: OK=&A016
A016  CD0B00 120  OK CALL &B
A019  C9     130  RET
```

programme : affichage

start : &A000   fin : &A019

longueur : 001A

0 erreurs

tableau de variables :
OK A016

XVII. SAUVEGARDE DE ROUTINES EN ASSEMBLEUR ET DE SECTEURS DE MEMOIRE.

Avec l'interprèteur BASIC formidable du CPC, vous êtes en mesure de sauvegarder des secteurs de mémoire. La syntaxe de ce type d'instructions est :

SAVE 'nomduprogramme',B,adressededepart,longueur

De cette manière, vous pouvez sauvegarder entre autres des routines en langage machine sur disquette ou sur cassette.

Exemple:

Le programme à sauvegarder se trouve de &A000 à &A013.

SAVE 'nom',B,&A000,&14

Faites attention à ce que la longueur ne soit pas trop courte. Dans l'exemple, la longueur donnée est de &14 octets, bien que le dernier octet ait le numéro &xx13. Il manque le dernier octet, qui est le RET qui permet le retour au BASIC.

Outre la possibilité de sauvegarder des routines en assembleur, vous pouvez aussi sauvegarder le contenu de l'écran.

SAVE 'SCREEN',B,&C000,&4000

Dans cet exemple, l'écran normal est sauvegardé à partir de l'adresse &C000 avec la longueur &4000 et sous le nom SCREEN.
Voici maintenant des possibilités intéressantes pour utiliser les routines screencopy, screenswap ou le transfert d'écran. Etant donné que le chargement d'une image complète prend quelques secondes, il est possible de charger l'écran dans le bloc 1 et de l'afficher rapidement avec :

LOAD 'SCREEN',&4000 : CALL screencopy

Adresse &0000 : RST 0-reset

L'appel de cette routine avec call 0 réinitialise l'ordinateur.

Adresse &0008 : RST &08-low jump

Cette routine se branche sur une adresse dans le système d'exploitation en ROM ou en RAM susjacente. Les bits 14 et 15 effectuent la sélection ROM/RAM. un bit à 1 signifie RAM, un bit à 0 signifie ROM. Le bit 14 indique la zone de mémoire inférieure (&0000-&3FFF) et le bit 15, la zone de mémoire supérieure (&C000-&FFFF)

Adresse &000C : JP (HL) avec sélection ROM/RAM

Branchement indirect à l'adresse située dans le registre HL. Les BITs 14 et 15 de HL ont le même rôle que pour RST &08.

Adresse &0010 : RST &10 : side call

Sert à appeler une routine dans une ROM extérieure.

Adresse &0018 : RST &18 : far call

Sert à appeler une routine n'importe où en ROM ou en RAM. Derrière l'instruction RST &18, se trouve l'adresse d'un vecteur qui doit contenir l'adresse de destination et le statut de la ROM/RAM.

Adresse &0020 : RST &20-RAM lam LD A,(HL)

L'accumulateur prend la valeur de l'adresse sur laquelle pointe HL.

Cette routine sélectionne toujours la RAM.

Adresse &0028 : RST &28-firm jump

Sert à l'appel d'une routine dans le système d'exploitation (firmware). L'adresse doit être précisée directement après l'instruction.

Adresse &0030 : RST &30-user restart

Cette routine sert pour certains programmes. Vous trouverez un exemple d'utilisation de RST &30 au chapitre 23.

Adresse &0038 : RST &38-interrupt pointer

Les routines d'interruption sont appelées par cette adresse

Adresse &BB06 : KM (keymanager)-wait char

Le code ASCII de la touche pressée est mis dans l'accu.

Adresse &BB15 : KM EXP buffer

Cette routine établit la zone RAM qui sera utilisée pour le tableau de redéfinition de touches. Etant donné que cette zone est un peu courte, elle peut être étendue avec cette routine. Pour cela, on met l'adresse de début dans le registre DE et la longueur du buffer dans le registre HL. Si on a donné une longueur suffisante, chaque touche peut être définie avec une chaîne faisant jusqu'à 32 caractères. Les vieilles données seront effacées par le fonctionnement de cette routine.

Adresse &BB24 : KM get joystick

Le registre H contient l'état du joystick 1, le registre L celui du deuxième.

Adresse &BB39 : KM set repeat

La fonction repeat est mise en route avec le numéro inscrit dans l'accu, si b<>0. Si b=0, elle est stoppée.

Adresse &BB3C : KM get repeat

Le flag Z est mis à 1 lorsque le numéro de l'accu n'a pas l'état pour le repeat, sinon la flag est mis à 0.

Adresse &BB3F : KM set delay

Numéro de la touche : A, temps de temporisation dans le registre H, vitesse de répétition dans le registre L.

Adresse &BB42 : KM set delay

idem

Adresse &BB5A : TXT output

Donne la valeur de l'accu en fonction du caractère sur l'écran.

Adresse &BB60 : TXT read char

Lit un caractère sur l'écran. La position du caractère est mise dans HL (h=ligne/L=colonne). Si un caractère est reconnu, le carry est mis A 1 et le code ASCII du caractère est chargé dans l'accu.

Adresse &BB6C : TXT clear window

Efface la fenêtre courante.

Adresse &BB75 : TXT set cursor

H/L correspond à la ligne/colonne

Adresse &bb78 : TXT get cursor

Adresse &BB81 : TXT cursor on

Adresse &BB84 : TXT cursor off

Adresse &BB90 : TXT SET PEN

La couleur du pen est mis dans l'accu avec le numéro d'ink.

Adresse &BB93 : TXT get pen

Lit le numéro courant de PEN-INK dans l'accu.

Adresse &BB96 : TXT set paper (s.pen)

Adresse &BB99 : TXT get paper (s.pen)

Adresse &BB9C : TXT inverse

Adresse &BBA5 : TXT get matrix

L'adresse de la définition du caractère du code A est mise dans le registre HL. Le carry est mis à 1 quand il s'agit d'un caractère défini par l'utilisateur.

Adresse &BBA8 : TXT set matrix

La matrice située à partir de l'adresse HL est chargée dans la matrice du caractère définissable par l'utilisateur avec le code A.

Adresse &BBC0 : GRA move absolute

Le registre DE contient la coordonnée X et le registre HL, la coordonnée Y.

Adresse &BBC3 : GRA move relative

Le registre DE contient la coordonnée X relative et le registre HL, la coordonnée Y relative.

Adresse &BBC6 : CRA ask cursor

cf move

Adresse &BBC9 : GRA set pen A=ink

Adresse &BBE1 : GRA get pen A=ink

Adresse &BBE4 : GRA set paper A=ink

Adresse &BBE7 : GRA get paper A=ink

Adresse &BBEA : GRA plot absolu

Adresse &BBED : GRA plot relative

DE: coordonnée relative X
HL: coordonnée Y relative

Adresse &BBF0 : GRA TXT absolu

DE : Coordonnée X
HL : coordonnée Y
Retour : A contient la couleur du point.

Adresse &BBF3 : GRA TXT relatif

Adresse &BBF6 : GRA draw line

La ligne est tracée du curseur graphique courant aux coordonnées cibles.

Adresse &BBF9 : GRA draw line relative

Adresse &BBFC : GRA write char

Le caractère A est écrit à la position du curseur graphique.

Adresse &BC0B : SET location

L'adresse courante du caractère gauche en haut de l'écran est donnée. A contient l'octet de poids fort de l'adresse de base (normalement &C0) et HL la différence entre l'adresse de base et l'adresse de départ.

Adresse &BC0E : SCR mode A

Le mode d'écran correspondant au contenu de l'accu est mis en route.

Adresse &BC11 : SCR GET mode

Le numéro de mode courant est chargé dans l'accu. En outre, les flags sont modifiés :

MODE 0 : C
MODE 1 : Z
MODE 2 : NC

Adresse &BC14 : SCR clear screen with ink=0

Adresse &BC1A : SCR CHAR POS

affichage : H-ligne/L-colonne
retour : HL : adresse du premier octet
B : largeur du caractère en octets(1,2 ou 3)

Adresse &BC1D : SCR DOT POS

affichage : DE : coordonnées X/ HL : coordonnées Y
retour : HL : adresse de l'écran
C : masque de bits qui ne tient compte que du point concerné.
B : nombre de points -1 par octet (1,2 ou 3)

Adresse &BC20 : SCR next byte

L'adresse d'écran HL augmentée d'un octet.

Adresse &BC23 : SCR prev byte

L'adresse d'écran HL est déplacée d'un octet vers la gauche.

Adresse &BC26 : SCR next line

L'adresse d'écran HL est mise sur la ligne suivante.

Adresse &BC29 : SCR prev line

L'adresse d'écran HL est mise sur la ligne précédente.

Adresse &BC32 : SET ink color

A : numéro d'INK; B et C couleurs.

Adresse &BC35 : SCR get ink color (comme au dessus)

Adresse &BC38 : SCR set bordercolor (B,C)

Adresse &BC3B : SCR GET bordercolor (B,C)

Adresse &BC3E : SCR set flash time

H : temps pour la première couleur
L : temps pour la deuxième couleur

Adresse &BC41 : SCR get flash time

Adresse &BC5f : SCR ligne à angle droit

A : ink, DE : début X, BC : fin X, HL : coordonnée Y.

Adresse &BC62 : CR ligne descendant

A :ink, DE : coordonnée X, BC : fin Y. HL : début Y.

Adresse &BC9B : CAS (ou. disc) catalog

Adresse &BCD1 : KL (kernal) log ext

Relie les extensions par RSX.

Adresse &BD0D : KL time please

Donne la valeur du timer dans HL et DE sur 4 octets.

Adresse &BD10 : KL time set

Adresse &BD2B : MC print char

Envoie la valeur de l'accu vers l'imprimante.

Adresse &BD2E : MC printer busy

Teste si l'imprimante est prête. Sinon, le flag carry est mis à 1.

Adresse &BD37 : jump restore

Donne à tous les vecteurs de saut éventuellement modifiés leur valeur initiale.

Adresse &BDD3 : write
Le contenu de l'accu est affiché à l'écran en tant que caractère ASCII

IXX. ROUTINES UTILES DE L'INTERPRETEUR BASIC

La liste d'adresse suivante doit être lue ainsi :

Adresse 1 pour le 464, adresse 2 pour le 664 et adresse 3 pour le 6128.

Adresse &C356 : &C3A0 : &C3A3 : PRINT

Envoie le caractère correspondant au contenu de l'accu sur le canal courant. Le numéro de canal est précisée en &AC21, &AC06, &AC06. L'affichage de notre programme XREF/DUMP par exemple peut être imprimée par poke &AC06,8: |XREF (464 : poke &AC21,8 :|XREF).

Adresse &C34E : &C398 : &C39B : line feed

Envoie un line feed sur le canal courant.

Adresse &C43C : &C472 : &C475 : BREAK test

Teste si ESC a été pressée. Si oui, la routine se branche sur le mode d'attente (comme en BASIC).

Adresse &CA94 : &CB55 : &CB58 : sortie d'erreur

En donnant le numéro de l'erreur dans l'accu (pour les 664/6128, dans le registre E), le message d'erreur correspondant s'affiche et le programme est arrêté.

Adresse &D5DB : &D619 : &D61C : accès au pointeur de tableau de lettres et de variables. La lettre de début de la variable à rechercher est donnée dans l'accu. Au retour, BC contient L'adresse de début et HL, l'adresse sur laquelle se trouve le pointeur de la première variable avec la lettre voulue.

Adresse &D6B3 : &D6EC : &D6EF : accès au pointeur de variables HL doit pointer sur le début d'une variable dans le programme, ou plus exactement sur le caractère de type qui définit toute variable. Si une variable existe dans le tableau, son adresse de début est donnée. Sinon, la variable est mise dans le tableau et sa nouvelle adresse de début est donnée.

Adresse &DD37 : &DE25 : &DE2A : CHR next

Teste si l'octet suivant (adresse HL+1) du programme BASIC est le même que celui qui est donné lors de l'appel de la routine. S'ils sont différents, un message 'syntax error' est affiché.

Adresse &DD3F : &DE2C : &DE31 : chrget

Cette routine sert à charger l'octet suivant. Pour cela, HL est augmenté. L'octet se trouvant à l'adresse spécifiée par HL est lu. S'il s'agit d'un caractère vide (code 32), le caractère suivant est lue, etc... Le premier caractère non vide rencontré est chargé dans l'accu. S'il s'agit d'un octet nul, le flag Z est mis à 1. La routine est utilisée pour la lecture de programmes BASIC pas à pas. L'octet nul correspond à la fin d'une ligne.

Adresse &DD51 : &DE3D : &DE42 : CHRGOT

Relit le dernier octet lu et teste si la fin d'une instruction a été atteinte. Si oui, C est mis à 1.

Adresse &DD55 : &DE41 : &DE46 : CHKvirgule

Teste si l'octet courant correspond à une virgule. Si oui, l'octet suivant est lu et le carry est mis à 1. Sinon, le flag de carry est effacé.

Adresse &E8FF : &E9B9 : &E9BE : BASIC DO routine BC

Cette routine parcourt le programme BASIC courant et se branche au début de chaque ligne à la routine désignée par l'adresse dans BC. A cette routine, la ligne est exécutée. Après son traitement, la ligne est quittée avec RET et la suivante est automatiquement chargée.

Adresse &E943 : &E9FD : &EA02 : SKIP COMMAND

Entre autres utilisée en relation avec la précédente, cette routine saute une instruction particulière.

Adresse &EE79 : &EF44 : &EF49 : PRINT HL décimal

Cette routine peut être utilisée pour l'affichage de numéros de lignes d'un programme BASIC. Il faut entrer le numéro de ligne dans le registre HL sur deux octets.

Adresse &F236 : &f2d5 : &f2DA : PRINT FAC

Cette routine donne la valeur contenue dans FAC sous forme décimale.

Adresse &FF4B : &FF6C : &FF6C : VAR FAC

Copie la valeur d'une variable dans FAC. HL doit contenir l'adresse de la valeur.

Adresse &FF71 : &FF92 : &FF92 : TEST lettre

Le contenu de l'accu est interprété en ASCII et si possible transformé en majuscules. S'il ne s'agissait pas du code ASCII d'une lettre, le carry est mis à 1.

Adresse &CFEE : &D058 : &D055 : OPERAND MISSING

Adresse &C205 : &CB30 : &C210 : IMPROPER ARGUMENT

XX. COMPATIBILITE ENTRE LES TROIS CPC

Dans ce chapitre, nous allons indiquer rapidement les différences et les ressemblances importantes entre les ordinateurs CPC 464, CPC 664 et CPC 6128, en ce qui concerne la programmation en langage machine. On peut affirmer que les ROMs des trois ordinateurs sont très proches.

Malheureusement cela signifie seulement que presque toutes les routines existent sous la même forme dans les trois appareils. Le hic est que toutes les routines ou presque ont une adresse de branchement différente. Heureusement, ce n'est pas le cas de la plupart des vecteurs de branchement en RAM. C'est vrai
en particulier pour les vecteurs:

- HIGH KERNEL
  Adresses &B900 à &B920
- KEY manager
  Adresses &BB00 à &BB4D
- TEXT pack
  Adresses &BB4E à &BBB9
- GRAphic pack
  Adresses &BBBA à &BBFE
- SCReen pack
  Adresses &BBFF à &BC64
- CAS (DISC) manager
  Adresses &BC65 à &BCA6
- SOUND pack
  Adresses &BCA7 à &BCC5
- KERNEL (low)
  Adresses &BCC8 à &BD12
- MaCHine pack
  Adresses &BD13 à &BD35
- comme le vecteur JUMP RESTORE &BD37

Malheureusement, nous n'avons pas pu tester toutes les routines. Cependant, nous n'avons pas encore trouvé de routine parmi celles que nous vous donnons au dessus qui ne fonctionne pas avec les trois CPC.

Après le vecteur JUMP RESTORE à l'adresse &BD37, c'en est fini de la compatibilité.

Dans le 464, ce sont les appels de l'éditeur de ligne et les branchements aux routines arithmétiques qui suivent. Pour le 664 et 6128, il y a quelques nouveaux vecteurs de branchement comme par exemple l'appel de la routine FILL. Le vecteur LINE EDITOR se trouve à l'adresse &BD3A, puis suivent des routines arithmétiques. Les vecteurs de JUMP (vecteurs qui ne doivent être appelés qu'à la mise sous tension) de l'adresse &BCCD à l'adresse &BDF3 sont les mêmes pour tous les ordinateurs.

Si vous voulez vous brancher directement sur n'importe quelle routine, comme les routines BASIC, les adresses des trois appareils sont différentes, mais il y a peu de différence entre les 664 et 6128.

Une autre différence importante est que les indirections de l'interpréteur BASIC du CPC 464 ne se retrouvent pas dans les 664 et 6128. Par conséquent, toutes les adresses système ont été déplacées (peek et poke). Mais ces adresses reste les mêmes pour la plupart en ce qui concerne les 664 et 6128.

En résumé :

Pour la programmation, il est conseillé d'utiliser autant que possible les vecteurs que nous avons indiqué. Si ce n'est pas possible, il sera difficile d'éviter d'écrire trois versions différentes. Un test de la routine KL version qui reste toujours à la même adresse &B915 (heureusement, sinon que faire?) vous donnera le numéro de la hig-ROM. H contiendra le numéro de la ROM (1 pour la ROM BASIC) et L le numéro de la version 0=464, 1=664, 2=6128.

A l'aide de cette routine, le programme affichage du chapitre sur le chargement de programme en langage machine, peut fonctionner sur les trois versions.

XXI. L'EXTENSION DU JEU D'INSTRUCTIONS AVEC RSX

On peut étendre les instructions du BASIC de différentes manières. Avec le 464, on peut étendre les routines de ROM dans le secteur de branchement de la RAM. Mais cette possibilité est relativement limitée, car on ne dispose que de 9 adresses libres. Avec le 6128, elles n'existent pas pour des raisons de gain de place. Pour tous les CPC, on peut utiliser la méthode vue précédemment qui permet l'extension avec ses propres instructions. Vous la connaissez tous, elle est utilisée par toutes les disquettes d'instructions AMSDOS.

Vous avez déjà sûrement appelé le mode CP/M avec |cpm si vous possédez un lecteur de disquettes. L'instruction CPM commence avec un '|'. Si vous entrez CPM sans '|', vous obtenez un message d'erreur. Si vous essayez |cpm sur un ordinateur sans floppy (464) vous obtenez le message 'unknow command'.

Il s'agit donc bien d'une instruction qui n'existe que lorsque le Floppy est branché. Ainsi c'est donc une extension.

Le | indique au système qu'il s'agit d'une extension. La méthode permettant de développer des instructions à partir d'un caractère spécial est prévue d'origine dans le CPC. Ce type d'extension est appelé RSX. RSX signifie 'resident system extension'. RSX est donc prévue pour lier de nouvelles instructions se trouvant dans des ROM d'extension au système. Il existe une telle ROM dans le floppy. Elle contient par exemple les instructions |CPM, |ERA. Nous pouvons nous aussi utiliser la méthode RSX pour utiliser nos propres routines en tant qu'instructions.

21.1. DOKE écriture de valeurs sur 2 octets en mémoire.

Nous allons expliquer l'utilisation de la méthode RSX avec l'instruction DOKE.
DOKE est une sorte de double poke. Avec POKE, on peut écrire une valeur allant de 0 à 255 en mémoire. Avec DOKE, on peut écrire une valeur allant de 0 à 65535. La valeur est divisée en octet de poids fort et en octet de poids faible. Avec DOKE, la transformation de nombreux paramètres système en RAM est facilitée.

La saisie de paramètres avec RSX est la même qu'avec l'instruction CALL.

L'instruction DOKE utilise deux paramètres: L'adresse à laquelle la valeur doit être sauvegardée et la valeur qui doit être mise à cette adresse. L'instruction a donc le format suivant:

|DOKE, adresse, valeur

Les paramètres sont :

A : contient le nombre des paramètres donnés.
Flags : le flag zéro=1 si aucun paramètre n'a été donné. Sinon =0.
B : contient 32. nombre du paramètre
DE : contient le dernier paramètre chargé.
IX : Adresse du dernier élément. L'avant dernier paramètre se trouve alors à l'adresse IX+2, le suivant à l'adresse IX+4, etc...

La routine DOKE apparaît ainsi:

```
100 ' LD L,(IX+2);     adresse donnée
110 ' LD H,(IX+3);     charger dans reg. HL
120 ' LD (HL),E;       mémorisation de l'octet de poids faible
130 ' INC HL;          adresse à l'octet de poids fort
140 ' LD (HL),D;       mémorisation de l'octet de poids fort
150 ' RET;             fini !
```

Avec CALL &A000,adresse,valeur vous pouvez appeler cette routine DOKE, à condition bien entendu qu'elle soit chargée à l'adresse &A000.

Essayez: CALL &A000,...,...

C'est donc une routine DOKE qui fonctionne. Mais l'appel par CALL est assez ennuyeux. Il est préférable de pouvoir appeler DOKE par DOKE. Pour cela, il faut construire une petite routine de liaison. Le traitement principal est effectué par la routine "LOGTEXT" du système d'exploitation qui crée la liaison. Mais il faut donner certaines valeurs à logtext.

Pour relier DOKE, le système doit avoir les informations suivantes:

1) Nom de l'extension
2) Adresse de la routine
3) Adresse des 4 octets inutilisés qui seront utilisés pour le traitement interne de la routine.

L'adresse des quatre octets système est chargée dans le registre HL avant la mise en route. En outre, BC prend l'adresse de départ du tableau dans lequel sont mémorisées les informations suivantes.

Ce tableau contient d'abord l'adresse d'un second tableau qui contient le nom ou les noms des routines à relier. Après l'adresse du second tableau se trouve une instruction de branchement qui appelle la ou les routines à relier. S'il faut relier plusieurs routines, les instructions de branchement et les noms des routines doivent correspondre.

Le second tableau contient donc les noms des routines. Pour celà, afin de séparer les noms, le bit 7 de la dernière lettre d'un nom est mis à 1.

Voici notre exemple :

10' LD BC,RSXTAB; adresse du premier tableau
20' LD HL,SYSBYT; adresse des octets système
30' CALL &BCD1; routine logtext
40' RET; liaison finie
50' RSXTAB DW NAMTAB; le tableau 1 contient comme première adresse l'adresse de départ du second tableau

```
60'   JP START; puis le branchement à la routine
70'   NAMTAB DM "DOK"; tableau de nom
80'   DB &C5; =ASC("e")+128 bit 7 à 1
90'   DB O; octet nul=fin du tableau
100'  START LD... la routine commence ici
```

Après l'assemblage du programme, l'instruction DOKE est disponible avec un CALL &A000. DOKE est donc une extension qui peut être utilisée en mode direct ou avec un programme.

Faites attention à ce que la routine de liaison ne soit appelée qu'une seule fois. Les quatre octets système sont utilisés entre autres pour pointer sur d'autres extensions. Si vous initialisez la routine une deuxième fois, le pointeur qui indique le second tableau RSX doit logiquement pointer sur le même tableau.

Celà peut entraîner une boucle sans fin. Le petit programme qui suit évite que la routine puisse être appelée une deuxième fois car il écrit un RET au début de la routine.

```
.COPYRIGHT 1985 MICRO-APPLICATION.
.DAMS *.*.

                    ;
                            ORG   #A000
                    INIT
#A000 010FA0                LD    BC,RSXTAB
#A003 2119A0                LD    HL,SYSBYT
#A006 CDD1BC                CALL  #BCD1
#A009 3EC9                  LD    A,#C9
#A00B 3200A0                LD    (INIT),A
#A00E C9                    RET
#A00F 14A0          RSXTAB  DEFW  NAMTAB
#A011 C31DA0                JP    START
                    ;
#A014 444F4BC5      NAMTAB  DEFB  "D","O","K",128+"E"
#A018 00                    DEFB  0
                    SYSBYT  DEFS  4
                    ;
                    START
#A01D DD6E02                LD    L,(IX+2)
#A020 DD6603                LD    H,(IX+3)
#A023 73                    LD    (HL),E
#A024 23                    INC   HL
#A025 72                    LD    (HL),D
#A026 C9                    RET

Pass 2: 0 Errors

End of code:#A027
```

21.2 RPEEK lecture à volonté de la RAM ou de la ROM

Le listing qui suit montre comment utiliser la fonction @ en relation avec RSX ou CALL.
Dans notre cas, l'adresse de départ de la valeur est assignée à une variable integer avec @ et chargée dans un programme en langage machine. La valeur assignée est écrite à l'adresse donnée et peut alors être utilisée en BASIC avec la variable désirée.

Le programme qui suit implémente une instruction PEEK étendue avec RSX. Avec cette nouvelle instruction, il est possible de lire des adresses aussi bien en RAM qu'en ROM ou sur des ROM d'extension.

L'instruction a le format :

RPEEK, variableinteger, adresse, statut

On peut écrire par exemple :

RPEEK, W%,&C000,-7

La valeur de la première adresse de la ROM disquette est chargée dans la variable W%. Il est important d'avoir utilisé au moins une fois W% auparavant, autrement W% donnerait le message d'erreur "improper argument". Le statut est défini ainsi :

1 : RAM
2 : ROM
0,-1,-2 à -251 sélectionnent les ROM d'extension.

Si vous utilisez d'autres valeurs pour le statut, vous obtiendrez le message "improper argument".

```
A000            10                  ; extension RSX
A000            20
; ":RPEEK,@Intvariable,Adresse,Statut"
A000            30                  ; Statut   1 :RAM
A000            40                  ;          2 :ROM
A000            50
;          0,-1,...,-251 :Expansionsrom
A000            60
A000            70  AOPMIS EQU  &d055
; 464 : &cfee / 664 : &d058
A000            80  AIMPAR EQU  &c21d
; 464 : &c205 / 664 : &cb50
A000 010000     90  INIT   LD   bc,rsxtab
A003 210000    100         LD   hl,sysbyt
A006 CDD1BC    110         CALL &bcd1
A009 3EC9      120         LD   a,&c9
A00B 3200A0    130         LD   (init),a
A00E C9        140         RET
**** Ligne  90  : RSXTAB=&A00F
A00F 0000      150  RSXTAB DW   namtab
A011 C30000    160         JP   start
**** Ligne  150  : NAMTAB=&A014
A014 52504545 170  NAMTAB DM   "RPEE"
A018 CB        180         DB   &cb
A019 00        190         DB   0
**** Ligne  100  : SYSBYT=&A01A
A01A           200  SYSBYT DS   4
A01E DF        210  OPMIS  RST  &18
A01F 0000      220         DW   vek1
A021 C9        230         RET
**** Ligne  220  : VEK1=&A022
A022 55D0      240  VEK1   DW   aopmis
A024 FD        250         DB   253
A025 DF        260  IMPARG RST  &18
A026 0000      270         DW   vek2
A028 C9        280         RET
**** Ligne  270  : VEK2=&A029
A029 1DC2      290  VEK2   DW   aimpar
```

```
A02B FD         300           DB    253
**** Ligne  160  : START=&A02C
A02C FE03       310   START   CP    3 ; deux parametres ?
A02E 20EE       320           JR    nz,opmis
; non, alors Operand missing
A030 7A         330           LD    a,d
A031 FE00       340           CP    0
A033 28FE       350           JR    z,noexro
; Statut>0, alors pas de ROM d'extension
A035 FEFF       360           CP    &ff ; nombre<255 ?
A037 20EC       370           JR    nz,imparg
; oui, alors improper Argument
A039 7B         380           LD    a,e
A03A ED44       390           NEG
A03C FEFC       400           CP    252
A03E 30E5       410           JR    nc,imparg
A040 18FE       420           JR    setsta
**** Ligne  350  : NOEXRO=&A042
A042 7B         430   NOEXRO  LD    a,e
A043 FE03       440           CP    3
A045 30DE       450           JR    nc,imparg
A047 C6FE       460           ADD   a,254
A049 30FE       470           JR    nc,setsta
A04B D604       480           SUB   4
**** Ligne  420  : SETSTA=&A04D
**** Ligne  470  : SETSTA=&A04D
A04D 320000     490   SETSTA  LD    (status),a
A050 DD6E02     500           LD    l,(ix+2)
A053 DD6603     510           LD    h,(ix+3)
A056 DF         520           RST   &18
A057 0000       530           DW    vektor
A059 DD6E04     540           LD    l,(ix+4)
A05C DD6605     550           LD    h,(ix+5)
A05F 77         560           LD    (hl),a
A060 97         570           SUB   a
A061 23         580           INC   hl
A062 77         590           LD    (hl),a
A063 C9         600           RET
**** Ligne  530  : VEKTOR=&A064
A064 0000       610   VEKTOR  DW    debut
```

```
**** Ligne  490  : STATUS=&A066
A066 FD        620  STATUS DB   253
**** Ligne  610  : DEBUT=&A067
A067 7E        630  DEBUT  LD   a,(hl)
A068 C9        640         RET

Programme :rpeek
Debut : &A000    Fin : &A068
Longueur : 0069
 0  Erreur
Table de variables :
AOPMIS D055  AIMPAR C21D  INIT   A000  RSXTAB A00F
NAMTAB A014  SYSBYT A01A  OPMIS  A01E  VEK1   A022
IMPARG A025  VEK2   A029  START  A02C  NOEXRO A042
SETSTA A04D  VEKTOR A064  STATUS A066  DEBUT  A067
```

```
10 FOR i=&A000 TO &A068
20 READ a$:w=VAL("&H"+a$)
30 s=s+w:POKE i,w:NEXT
40 IF s<> 13319 THEN PRINT"Erreur dans les Datas":END
50 '464: IF s<> 13447 THEN ...
60 '664: IF s<> 13382 THEN ...
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,1A,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,14
100 DATA A0,C3,2C,A0,52,50,45,45
110 DATA CB,00,20,A0,0F,A0,DF,22
120 DATA A0,C9,55,D0,FD,DF,29,A0
130 '464:..,..,ee,cf,..,..,..,..
140 '664:..,..,58,d0,..,..,..,..
150 DATA C9,1D,C2,FD,FE,03,20,EE
160 '464:..,05,c2,..,..,..,..,..
170 '664:..,50,cb,..,..,..,..,..
180 DATA 7A,FE,00,28,0D,FE,FF,20
190 DATA EC,7B,ED,44,FE,FC,30,E5
200 DATA 18,0B,7B,FE,03,30,DE,C6
210 DATA FE,30,02,D6,04,32,66,A0
220 DATA DD,6E,02,DD,66,03,DF,64
230 DATA A0,DD,6E,04,DD,66,05,77
240 DATA 97,23,77,C9,67,A0,FD,7E
250 DATA C9
```

XXII. LA CONSTRUCTION DE VARIABLES DE CHAMP

Dans ce chapitre, nous allons parler un peu de la sauvegarde des variables de champ. Nous vous présenterons comme exemple pratique un programme qui calcule la somme de tous les éléments en langage machine.

Les variables de champ sont sauvegardées comme les variables normales après le programme BASIC dans un tableau. L'adresse de départ du tableau est à &AE87 (664/6128: &AE68). Comme les variables de champ prennent beaucoup de place, il faut réserver le champ avec l'instruction DIM. Les variables de champ sont ordonnées en fonction de leur adresse de départ et de leur première lettre dans le tableau de chaînage (comme les variables normales).

La structure du tableau des variables de champ est la suivante :

D'abord il y a deux octets qui contiennent l'adresse de chaînage. Suit le nom de la variable au code ASCII de la dernière lettre duquel est ajouté 128=&80, ce qui met le bit 7 de ce code à 1. Puis suit le caractère de type. Le tableau des variables de champ ne diffère pas de celui des variables normales pour ce qui vient d'être vu.

Puis, on trouve les informations suivantes qui sont sauvegardées en tant que nombres sur deux octets. Il y a le nombre d'indices du champ, puis pour chaque indice sa taille maximale.

Après ces informations, se trouve la liste de toutes les valeurs du champ. Pour un champ INTeger, chaque valeur est donnée sur deux octets. Pour les variables REAL, les valeurs sont écrites sur 5 octets. Pour les champs de chaînes de caractères, le descripteur de chaîne qui fait 3 octets est mémorisé.

Résumé de la structure :

2 octets	adresse de chaînage
n octets	nom
1 octet	descripteur de type
2 octets	longueur des informations qui suivent
1 octet	nombre d'indices
tous les 2	octets: valeur maximale de chaque indice
m octets	valeur de chaque élément longueur=2 pour INTeger longueur=3 pour chaîne longueur=5 pour REAL

Le programme qui suit utilise cette structure pour donner la somme de tous les éléments d'un champ. Pour le format de l'instruction référez vous au listing en assembleur qui suit.

```
A000            10
; somme des elements d'un ensemble
A000            20
; donne la somme de toutes les valeurs
A000            30                  ; d'un ensemble de variables
A000            40
A000            50
; Format: "call &a000,@<nomdevariable(indices 0,..)>,nombre
Indices,@<Variable pour le resultat>
A000            60
A000            70          ORG   &a000
A000 DF         80          RST   &18
A001 0000       90          DW    vektor
A003 C9        100          RET
**** Ligne  90  : VEKTOR=&A004
A004 0000      110  VEKTOR DW    sumar
A006 FD        120          DB    253 ; UROM on
A007           130
A007           140
; Adr. fuer 6128 ;  464  ,  664
A007           150  IMPARG EQU   &c21d
; &FA9C , &cb50  improper arg.
A007           160  TYPMIS EQU   &ff62
; &FF40 , &ff62  type mismatch
A007           170  VARHND EQU   &ff8c
; &FF66 , &ff87  Variable (hl) apres (de)
A007           180  VARFAC EQU   &ff6c
; &ff4b , &ff6c  Variable (hl) apres FAC
A007           190  CPHLDE EQU   &ffd8
; &ffb8 , &ffb8  CP hl,de
A007           200  READHD EQU   &bd7c
; &bd58 , &bd79  arithm. reels (hl)+(de)
A007           210  FAC    EQU   &b0a0 ; &b0c2 , &b0a0
A007           220  TYP    EQU   &b09f ; &b0c1 , &b09f
A007           230
**** Ligne  110  : SUMAR=&A007
A007 D5        240  SUMAR  PUSH de
A008 210000    250          LD    hl,FAC1 ; mettre les cinq
```

```
A00B 0605               LD   b,5 ; octets de FAC1
A00D 3600         NEXT  LD   (hl),0 ; a zero
A00F 23                 INC  hl
A010 10FB               DJNZ next
A012 DD6E04             LD   l,(ix+4)
A015 DD6605             LD   h,(ix+5)
; Adresse du dernier element apres HL
A018 DD7E02             LD   a,(ix+2) ; nombre d'indices
A01B 47                 LD   b,a
A01C E5                 PUSH hl
; Adresse du premier element
A01D 87                 ADD  a,a ; soustraire
A01E C601               ADD  a,1 ; le nombre d'indices*2
A020 1600               LD   d,0 ; de HL
A022 5F                 LD   e,a
; tous les deux octets pour chaque indice
A023 B7                 OR   a ; effacer le carry
A024 ED52               SBC  hl,de
; HL pointe sur le nombre
A026 7E                 LD   a,(hl) ; d'indices
A027 B8                 CP   b ; comparaison
A028 C21DC2             JP   nz,imparg
; quand le nombre d'indices est mauvais: "improper Argument"

A02B 2B                 DEC  hl ; Longueur de l'ensemble
A02C 2B                 DEC  hl ; saut
A02D 2B                 DEC  hl ; HL pointe alors
A02E 7E                 LD   a,(hl)
; sur le caractere de controle du type
A02F C601               ADD  a,1
A031 FE03               CP   3 ; si chaine de caracteres
A033 CA62FF             JP   z,typmis
; alors Type mismatch
A036 4F                 LD   c,a
A037 329FB0             LD   (TYP),a
A03A 23                 INC  hl
A03B 5E                 LD   e,(hl) ; additionner
A03C 23                 INC  hl
; la longueur de l'ensemble
A03D 56                 LD   d,(hl) ; en octets a HL
```

```
A03E 19        570          ADD  hl,de
; addition=fin de l'ensemble
A03F E3        580          EX   (sp),hl
; echange du debut et de la fin
A040 0600      590          LD   b,0
A042 F3        600          DI    ; car c'est plus rapide
A043 E5        610   WEITER PUSH hl
; Adresse de l'element suivant
A044 C5        620          PUSH bc ; Typ dans c
A045 CD6CFF    630          CALL varfac ; var (hl) apres FAC
A048 21A0B0    640          LD   hl,fac
A04B 110000    650          LD   de,fac1
A04E FE05      660          CP   5 ; Reel ?
A050 28FE      670          JR   z,real
A052 7E        680          LD   a,(hl)
A053 23        690          INC  hl ; charger la valeur de
A054 66        700          LD   h,(hl) ; L'element courant
A055 6F        710          LD   l,a
A056 EB        720          EX   de,hl
A057 7E        730          LD   a,(hl) ; charger la valeur
A058 23        740          INC  hl ; de la somme
A059 66        750          LD   h,(hl)
A05A 6F        760          LD   l,a
A05B 19        770          ADD  hl,de
A05C 220000    780          LD   (FAC1),hl ; FAC1=hl
A05F 18FE      790          JR   break
**** Ligne  670  : REAL=&A061
A061 EB        800   REAL   EX   de,hl
A062 CD7CBD    810          CALL readhd ; real (hl)=(hl)+(de)
**** Ligne  790  : BREAK=&A065
A065 C1        820   BREAK  POP  bc ; Typ dans c
A066 E1        830          POP  hl ; HL pointe sur l'adresse
A067 09        840          ADD  hl,bc ; suivante
A068 D1        850          POP  de ; adresse de fin
A069 D5        860          PUSH de
A06A CDD8FF    870          CALL cphlde
; Comparaison de HL avec DE
A06D 38D4      880          JR   c,weiter
A06F FB        890          EI    ; fin de la somme
A070 D1        900          POP  de
```

```
A071 210000   910          LD   hl,FAC1
A074 D1       920          POP  de ; copier le resultat
A075 CD8CFF   930          CALL varhnd ; dans l'adresse cible
A078 C9       940          RET
**** Ligne  250  : FAC1=&A079
**** Ligne  650  : FAC1=&A079
**** Ligne  780  : FAC1=&A079
**** Ligne  910  : FAC1=&A079
A079           950  FAC1   DS   5

Programme :somme
Debut : &A000    Fin : &A07D
Longueur : 007E
 0  Erreur
Table de variables :
VEKTOR A004   IMPARG C21D   TYPMIS FF62   VARHND FF8C
VARFAC FF6C   CPHLDE FFD8   READHD BD7C   FAC    B0A0
TYP    B09F   SUMAR  A007   NEXT   A00D   WEITER A043
REAL   A061   BREAK  A065   FAC1   A079
```

```
10 REM chargeur BASIC pour 6128
20 REM programme somme des elements d'un ensemble
30 FOR i=&A000 TO &A07D
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 15294 THEN PRINT"erreur dans les Datas":END
70 PRINT"ok!":END
80 DATA DF,04,A0,C9,07,A0,FD,D5
90 DATA 21,79,A0,06,05,36,00,23
100 DATA 10,FB,DD,6E,04,DD,66,05
110 DATA DD,7E,02,47,E5,87,C6,01
120 DATA 16,00,5F,B7,ED,52,7E,B8
130 DATA C2,1D,C2,2B,2B,2B,7E,C6
140 DATA 01,FE,03,CA,62,FF,4F,32
150 DATA 9F,B0,23,5E,23,56,19,E3
```

```
160 DATA 06,00,F3,E5,C5,CD,6C,FF
170 DATA 21,A0,B0,11,79,A0,FE,05
180 DATA 28,0F,7E,23,66,6F,EB,7E
190 DATA 23,66,6F,19,22,79,A0,18
200 DATA 04,EB,CD,7C,BD,C1,E1,09
210 DATA D1,D5,CD,D8,FF,38,D4,FB
220 DATA D1,21,79,A0,D1,CD,8C,FF
230 DATA C9,06,08,0D,20,CC
```

```
10 REM somme des elements d'un ensemble pour 464
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15372 THEN PRINT"Erreur dans les Datas":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,9C,FA,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,40,FF,4F,32
140 DATA C1,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,4B,FF
160 DATA 21,C2,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,58,BD,C1,E1,09
200 DATA D1,D5,CD,B8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,66,FF
220 DATA C9,06,08,0D,20,CC
```

```
10 REM somme des elements d'un ensemble pour 664
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15346 THEN PRINT"Erreurs dans les Datas":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,50,CB,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,62,FF,4F,32
140 DATA 9F,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,6C,FF
160 DATA 21,A0,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,79,BD,C1,E1,09
200 DATA D1,D5,CD,D8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,87,FF
220 DATA C9,06,08,0D,20,CC
```

XXIII. POUR RENDRE LES PROGRAMMES EN LANGAGE MACHINE RELOGEABLES

23.1. Pourquoi relogeables ?

Toute personne qui a déja utilisé des programmes en langage machine et plus particulièrement des extensions d'instructions, qu'ils soient réalisés par elle-même ou par d'autres, connaît le problème: Chaque programme pris seul fonctionne sans difficultés. Mais si on veut combiner plusieurs programmes, c'est à dire les avoir en même temps dans la mémoire, les problèmes commencent. Un des problèmes principaux dans la combinaison d'extensions d'instructions tient à la gestion des vecteurs. Cela conduit au fait que l'extension chargée en dernier est liée à elle même par le vecteur de l'extension précédente. Par chance, ce problème n'existe pas sur le CPC grâce à l'instruction RSX. Le mécanisme RSX permet de combiner des instructions pratiquement à volonté. Mais il ne faut pas rétablir des vecteurs du système d'exploitation pour reconnaître les instructions nouvelles. La reconnaissance de nouvelles instructions est prévue dans le système d'exploitation.

L'autre problème qui résulte de la combinaison de programmes en langage machine ne vient pas du système d'exploitation mais tient à la particularité du langage machine. Un programme en langage machine est en principe lié à une zone mémoire pour laquelle il a été écrit. Il peut être mis à n'importe quel endroit de la mémoire mais il n'est utilisable qu'à la zone pour laquelle il a été fait. Cela vient de ce que beaucoup d'instructions font des adressages absolus. Pour les instructions qui adressent le programme lui même, l'effet évident est que le branchement n'est effectué correctement que lorsque le programme ou les données sont à la bonne place en mémoire. Etant donné que les extensions d'instructions se trouvent tout en haut de la mémoire afin de prendre le moins de place possible au BASIC, leur combinaison pose un problème compliqué : la nouvelle instruction chargée entre en collision avec l'ancienne et l'écrase.

La conséquence fatale est le plantage de l'ordinateur ou des défauts dans le fonctionnement de la première instruction.

23.2. Le Fonctionnement global des relocations

Le mot magique pour éviter les difficultés décrites jusqu'ici pour la combinaison de programmes s'appelle 'relocation'. Derrière ce mot se cache une technique qui rend un programme indépendant des adressages sur lui même et partiellement indépendant de l'adressage absolu. Pour arriver à cela, il y a deux possibilités :

La première possibilité est basée sur l'utilisation de fichiers temporaires qui contiennent un code spécial de chargement qui lui même contient des informations importantes pour la relocation. Ce code de chargement n'est pas un programme en langage machine. Il faut d'abord le mettre dans la zone mémoire prévue au moyen d'un programme de chargement spécial. Dans cette zone mémoire, il sera automatiquement transformé en programme. Le programme utilisé pour cette opération n'est pas un petit programme de chargement mais il s'agit d'un 'linker'. Un linker ne sert pas seulement à charger des programmes mais aussi à relier des modules de programmes à un programme principal (en anglais 'to link'='relier'). L'assembleur MACRO80 de la firme microsoft, par exemple fonctionne de cette manière. Le linker qui en fait partie permet outre la mise en route du programme en tant que code exécutable sur disquette ou en mémoire, la liaison automatique de routines d'une bibliothèque. Même la liaison entre un module-programme et une partie ou des données d'une autre module n'est pas un problème. Les programmes sont désignées par des label globaux qui sont gérés par le linker.

En ce qui concerne les possibilités offertes, cette solution est la meilleure. Elle offre des possibilités dont la flexibilité et les avantages ne sont limités que par la place mémoire, la rapidité et les possibilités d'interruption de l'ordinateur. Des programmes en langage machine importants peuvent être structurés et rendus modulaires avec cet utilitaire. Mais ils ne sont pas particulièrement lisibles et adaptables.

Il est possible par exemple de mettre le programme et les données dans des zones mémoires séparées comme on veut, ce qui est utile lorsqu'on veut programmer une EPROM. Les avantages qui précèdent ne sont pas atteints facilement : il s'agit d'un assembleur spécial qui crée le code de chargement. Cet assembleur n'est pas seulement plus complet qu'un assembleur normal, mais en raison de ses possibilités, il est difficile de l'utiliser dans toutes ses capacités. Le linker lui même n'est pas vraiment un petit programme.

La seconde possibilité peut être utilisée avec un assembleur tout à fait normal. Le programme est mis à une adresse quelconque avec l'assembleur. (lors du test du programme, il faut une adresse à laquelle le programme puisse fonctionner). Puis le travail commence : Le programme qui doit être relogeable doit être parcouru par un programme de chargement spécial qui demande l'adresse de base ou qui la retrouve et qui calcule les adresses absolues du programme et les change avant de mettre en route le programme. Le programme de chargement ne change pas beaucoup et peut être écrit en BASIC. Mais il faut écrire le tableau des adresses correspondantes et l'adapter à chaque programme. Ce tableau est déjà immense pour des programmes moyens et il grandit considérablement avec la taille du programme. On pourrait imaginer de créer automatiquement ce tableau avec par exemple un programme qui tirerait ses informations de commentaires spéciaux. Mais le développement d'un programme de ce type et sa manipulation risque d'être assez difficile.

Il y a encore une autre méthode qui est basé sur l'expérimentation: un relocalisateur automatique' désassemble le programme à traiter et prend en main la modification des adresses sans tenir compte des instructions qui contiennent des adressages absolus vers le programme (instruction de chargement sur 16 bits, branchements absolus et instructions de chargement avec adressage étendu). Cette méthode est très simple mais dès que le programme contient un tableau d'adresses qui concernent le programme lui-même, elle ne fonctionne plus. De plus, une valeur qui suit une instruction de chargement sur 16 bits n'est pas obligatoirement une adresse, sauf si elle correspond à une adresse du programme.

Les valeurs de compteurs et autres compteurs seraient de même modifiées par un tel 'relocalisateur'. Pour ces raisons, ce serait un hasard si un tel programme pouvait fonctionner.

23.3 Mais il y a plus simple !

Les deux premières méthode de relocation décrites sont largement utilisées, ce qui n'est pas vraiment le cas de la troisième méthode. Mais ces deux méthodes ont un défaut: la première exige de se procurer un logiciel pas particulièrement bon marché et d'étudier une nouvelle méthode de programmation en assembleur. La seconde demande beaucoup de travail.

La méthode que nous voulons vous présenter ici diffère sensiblement des précédentes. Vous allez apprendre une méthode de relocation qui avec une taille minimum (le programme présenté ici tient sur 42 octets !) prendra en main pratiquement tous les problèmes de relocation.

Le principe sur lequel cette méthode est basée est tout simplement génial : le problème de la relocation ne se poserait même pas s'il y avait des instructions équivalentes aux instructions de branchement relatif pour les JP et les CALL (avec branchement sur 16 bits afin d'avoir accès à toute la mémoire), ainsi que pour les instructions de chargement avec adressage étendu ou sans extension. Nous avons donc pensé: si cette instruction n'existe pas, il suffit de l'inventer ! Mais la création d'une nouvelle instruction n'est pas chose courante, aussi nous allons la décrire précisément. Pour construire notre nouvelle instruction, nous utilisons une méthode que les constructeurs du Z80 ont crée : la méthode du préfixage. Elle est basé sur le fait qu'un codeOP défini prend une signification en fonction d'un préfixe qui lui est assigné. Ainsi, une instruction qui concerne le registre HL pourra concerner le registre IX avec le préfixe &DD ou le registre IY avec le préfixe &FD. Avec notre préfixe, nous ferons en sorte qu'une instruction avec une opérande sur 16 bit ne prenne pas cette opérande de façon absolue mais relative à la position du PC.

De cette manière, des parties de programme ou bien des données pourront être reliées à l'adresse de l'instruction donnée ce qui permettra de déplacer l'adresse absolue du programme.

Naturellement, nous ne changerons pas simplement la fonction du processeur Z80. Au lieu de cela, nous utiliserons un petit programme (42 octets) qui sera appelé lors de la rencontre de notre préfixe et qui réagira en conséquence. En tant que préfixe, nous utiliserons une instruction restée libre pour l'utilisateur RST &30 (effectue CALL &0030). Un simple ordre CALL aurait des inconvénients en ce qui concerne la rapidité de traitement et la place mémoire. L'avantage de l'instruction RST tient au fait qu'elle fonctionne sans adresse de branchement et n'utilise qu'un octet. L'instruction RST &30 est mise simplement avant l'instruction à interpréter . Avant d'exécuter l'instruction durant le programme, le processeur rencontre RST &30 et exécute le sous programme qui lui est assigné. Ce sous programme prend l'adresse de l'instruction considérée (Où se trouve l'instruction, elle se trouve à l'adresse de retour de RST située sur la pile), calcule à partir de cette adresse et de l'adresse de base (qui est l'adresse de retour) l'adresse effective. Il écrit ensuite cette adresse derrière l'instruction, efface le RST qui l'a appelé avec une instruction NOP et revient au programme pour exécuter l'instruction. Avec cette méthode, le programme est relié à sa place mémoire dès qu'il a été exécuté. Il est pratiquement très rare qu'un programme doive être déplacé au cours de son exécution. Dans ce cas, il faudrait un programme d'aide très compliqué et qui prendrait beaucoup de temps. Un tel programme devrait contrairement au notre reconnaître les instructions et les modifier au cas par cas. Notre programme se contente de donner une adresse correcte à une instruction et laisse le processeur l'exécuter. la rapidité de notre programme ne tient pas seulement au fait qu'il ignore le rôle de l'instruction succédant au RST mais aussi au fait qu'il n'est actif qu'à la première exécution de l'instruction. Ensuite se trouve après l'instruction traitée une adresse correcte et l'instruction peut être exécutée. Le seul temps de traitement en plus est celui pris par l'instruction NOP qui travaille très vite.

23.4 Le programme de relocation

Nous en avons fini avec la théorie. Voici le listing de notre programme. Dans la description des fonctions, les bases théoriques de la méthode seront sans doute plus claires.

```
0000  e3     relrst  ex  (sp),hl     ;sauvegarde de hl
                                      adresse de retour en hl
0001  d5             push de         ; sauvegarde de
0002  c5             push bc         ; sauvegarde bc
0003  f5             push af         ; sauvegarde af
0004  e5             push hl         ; sauvegarde adresse retour
0005  7e             ld a,(hl)       ; 1er octet de l'instruction
0006  fe ed          cp &ed          ; teste préfixe
0008  28 08          jr z,skppfx
000a  fe dd          cp &dd
000c  28 04          jr z,skppfx
000e  fe fd          cp &fd
0010  20 01          jr nz,nopfx
0012  23     skppfx  inc hl          ; saute préfixe
0013  23     nopfx   inc hl          ; hl pointe adresse champ
0014  5e             ld e,(hl)       ; affiche octet faible
0015  23             inc hl
0016  56             ld d,(hl)       ; charge octet fort
0017  2b             dec hl          ; hl pointe debut du champ
0018  eb             ex de,hl        ; pointeur adresse champ
                                       après de, affichage hl
0019  c1             pop bc          ; adresse retour en bc
001a  c5             push bc         ; retour sur pile
001b  09             add hl,bc       ; additionne adresse base
001c  eb             ex de,hl        ; pointeur adresse champ sur
                                       hl efface adr. après de
001d  73             ld (hl),e       ; eff. adresse octet faible
001e  23             inc hl
001f  72             ld (hl),d       ; octet fort dans adresse
                                       champ
0020  e1             pop hl          ; adresse retour ds hl
0021  2b             dec hl          ; hl pointe sur rst
0022  36 00          ld (hl),&00     ; rst remplace par nop
```

```
0024  23    inc hl        ; hl contient adr. retour
0025  f1    pop af        ; reprend af
0026  c1    pop bc        ; reprend bc
0027  d1    pop de        ; reprend de
0028  e3    ex (sp),hl    ; reprend hl, adresse retour
                            sur pile
0029  c9    ret
```

Le fait que l'adresse de base de cet assembleur soit &0000 signifie peut être selon vous que le programme ne fonctionnera jamais à cette adresse. Le secteur &0000 à &003f est absolument tabou car il contient les vecteurs système les plus importants et les plus utilisés. (entre autres par exemple le vecteur RST &30). Mais en fait, ce n'est pas un problème car il s'agit en l'occurence d'une adresse relative. Vous n'écrivez donc pas l'adresse absolue mais la distance avec une adresse de départ qui reste à déterminer. Il est courant de donner des adresses relatives avec des assembleurs relogeables. Il serait stupide de donner des adresses absolues alors que L'adresse de base n'est même pas encore connue. Si cela vous gène, inscrivez simplement un A à la place de chaque premier 0 des adresses. Cela ne change rien au programme sinon qu'il fonctionnera avec l'adresse de base &A000. Notre petit programme d'aide est relogeable car il ne contient aucune instruction qui fasse un adressage absolu sur lui même. Cela signifie qu'il peut être inscrit sans précaution particulière à n'importe quelle secteur de mémoire prévu pour les programmes. Nous allons maintenant vous donner plus d'explication pour trouver l'adresse de base effective.

Malgré sa petite taille, ce programme rempli parfaitement les conditions précisées.

Il reste une particularité du programme que nous n'avons pas encore évoquée: comme il s'agit pratiquement d'une extension des instructions du Z80, il faut faire attention à ce qu'aucun registre ne soit modifié. Ce programme ne pourra être utilisé universellement que s'il n'a aucune influence sur les registres.

Pour ces raisons, les registres qui seront utilisés sont sauvegardés au début du programme (PUSH) et repris de la pile à la fin du programme (POP). Enfin, une dernière particularité qui bien que sans importance dans la plupart des cas peut vous faire chercher pendant longtemps si il y a une erreur: étant donné que le système d'exploitation du cpc doit déconnecter la ROM pour avoir accès à l'instruction RST, cette ROM est mise hors service après l'appel de RST &30.

23.5 Le programme de chargement

Avant d'en venir à une description des fonctions avec un exemple, il nous faut répondre à deux questions importantes concernant le fonctionnement du programme : 'Comment le programme prend-t-il le format d'exécution ?' (le listing ne contient pas l'adresse de base, pour des raisons précises) et 'Comment se fait-il qu'on se branche sur le programme de relocation avec RST &30 ?'.

Répondons tout d'abord à la seconde question. Ormis le fait qu'il ne fait qu'un seul octet et qu'il soit exécuté plus rapidement, l'instruction RST &30 est équivalente à l'instruction CALL &0030. A la place de &0030 se trouvent en RAM 8 octets dont l'utilisateur peut se servir pour sa routine RST. Etant donné qu'aucune routine ne sera suffisamment courte pour tenir sur les 8 octets, il faut y inscrire un vecteur qui branchera sur la routine désirée. Il sera constitué d'une instruction de branchement (JP). Il suffit donc d'inscrire le code op correspondant en &0030 et l'adresse de branchement de notre programme en &0031 et &0032. Etant donné que nous utilisons l'adresse de départ de notre routine pour lui assigner le vecteur RST (elle est la même que l'adresse de base), nous en venons à la question suivante : comment trouve-t-on l'adresse de base et comment le programme s'y trouve-t-il ?

Lorsque notre sous programme est inscrit en mémoire, il nous apparaît qu'il est relogeable sans modification. Il est donc possible d'écrire un chargeur pour notre programme qui l'inscrive à la meilleure place, c'est à dire au sommet de la mémoire libre.

Pour charger le programme nous utilisons une variante du chargeur BASIC habituel. Compte tenu de la taille du programme, ce chargeur a été assez simple à écrire.

Le voici :

```
10 rem chargeur BASIC pour programme d'aide à la relocation ver.
   1.0.
20 rem (hr) 11/85
30 memory himem-42
40 for mptr=himem+1 to himem+42
50 read byte
60 poke mptr,byte
70 cs=cs+byte
80 next
90 if cs<>&h165c then print chr$(7);'test de somme: erreur!';
   hex$(cs):stop
100 rem établissement du vecteur rst
105 IF  HIMEM>32767  THEN  temp=HIMEM-65536
    ELSE  temp=HIMEM
110 POKE   &30,&C3:POKE  &31,(temp+1)
    AND  255:POKE  &32,(HIMEM+1)/256
120 print 'programme d'aide à la relocation chargé correctement
    en'; hex$(himem,4)
130 end
1000 data &he3,&hd5,&hc5,&hf5,&he5,&h7e,&hfe,&hed
1005 data &h28,&h08,&hfe,&hdd,&h28,&h04,&hfe,&hfd
1010 data &h20,&h01,&h23,&h23,&h5e,&h23,&h56,&h2b
1015 data &heb,&hc1,&hc5,&h09,&heb,&h73,&h23,&h72
1020 data &he1,&h2b,&h36,&h00,&h23,&hf1,&hc1,&hd1
1030 data &he3,&hc9
```

DESCRIPTION DU PROGRAMME :

30 : avec la mise en place de HIMEM. on réserve de la mémoire au sommet de la mev.

40-80 : le code machine est lu dans les datas et chargé en mémoire réservée. En ligne 70, la somme test est faite.

90 : comparaison de la somme test avec la somme étalon et message d'erreur avec arrèt du programme.

110 : positionnement du vecteur RST &30 sur l'adresse de début du programme.

120 : avec l'adresse de base, le résultat du chargement est donné.

130 : fin du programme. A la place de END, on peut écrire l'instruction NEW qui efface le chargeur après le chargement. Naturellement, dans ce cas, il est conseillé de sauvegarder le programme auparavant.

Ce chargeur BASIC fait le nécessaire pour la mise en place de notre programme : calcul de l'adresse de base (HIMEM-longueur du programme), la réservation de la place mémoire (MEMORY), le transfert du programme et la fixation des vecteurs. Il ne reste plus qu'à l'essayer...

23.6 Utilisation du programme de relocation

Quelles doivent êtres les caractéristiques d'un programme assembleur pour tester le programme de relocation ? Les adaptations à apporter à un programme en assembleur normal pour qu'il soit utilisable par notre programme de relocation sont minimes. Il suffit d'apporter les modifications suivantes :

* Chaque instruction qui utilise une adresse sur le champ de son opérande doit être précédée de RST &30.

* Dans les mêmes instructions, il faut fixer une adresse relative au PC sur 16 bits. Une adresse relative au PC ne donne pas la différence avec l'adresse de base du programme mais la distance par rapport à l'instruction considérée à comme pour l'instruction JR).

La construction de l'instruction RST dans un programme mémorisé ne pose pas de problème, mais qu'en est-il des adresses relatives au PC ?
Cette méthode d'adressage ne demande pas de connaissances particulières. L'utilisation d'adresses relatives au PC est très importante car le programme tire ses informations directement ou indirectement de la pile. Pour cette raison, l'inscription du programme à traiter dans une quelconque table d'adresses de base est inutile.

En donnant la distance sur 16 bits, nous avons également prévu un secteur d'adressage qui couvre la mémoire entière de l'ordinateur (pour le 6128 : seulement un banc). La fixation d'adresses relatives au PC dans un programme en assembleur est très simple. Il suffit d'ajouter à cette adresse le signe $.

Si vous appelez par exemple un sous programme qui se trouve à l'intérieur du programme à traiter avec l'instruction suivante (l'appel se fait sur une adresse du programme) :

```
CALL NUMOUT
```

Pour modifier cette instruction, il faut écrire :

```
RST &30
CALL NUMOUT-$
```

Le '-$' n'est pas une instruction magique et inconnue. L'assembleur permet d'avoir accès au compteur de programme et même à l'adresse de l'instruction sur la ligne de laquelle se trouve le $.

L'instruction

```
JP $
```

est une boucle sans fin.

Nous soustrayons donc l'adresse de l'instruction de l'adresse absolue à laquelle l'instruction doit avoir accès et nous obtenons l'adresse relative.

Remarque : L'accès au compteur de programme est possible dans la plupart des assembleurs. Le caractère $ est le plus courant. Mais il arrive que d'autres caractères soient utilisés, comme le '*' par exemple.

Une fois que toutes les instructions qui adressaient à l'intérieur du programme ont été traitées ainsi, le programme est prêt et il peut fonctionner à n'importe quelle place en mémoire, une fois que le programme de relocation a été installé.

23.7 Un exemple de programme relogeable

Nous allons maintenant vous démontrer l'utilisation du programme de relocation avec un exemple pratique. Ce que fait le programme ne va pas révolutionner le monde : il compte de 100 à 0 et retourne au BASIC.

Bien que le programme ne soit pas écrit de façon pratique, compte tenu de son caractère d'exemple, il propose des routines intéressantes. Le programme a été écrit de façon totalement indépendante du système d'exploitation (ormis la routine TXT OUTPUT) afin de montrer l'appel relatif de sous programmes. Il contient une propre routine de calcul décimal : la routine NUMOUT. Cette routine d'affichage appelle la routine DIV168 qui effectue une division binaire. Nous allons observer cette routine de plus près car elle est un exemple clair d'une routine arithmétique compliquée.

Voici d'abord le listing du programme :

```
.COPYRIGHT 1985 MICRO-APPLICATION.
.DAMS *.*.

                    ;
                    ; Programme de demo pour RELRST
                    ;
                            ORG   #9000
                    ;
                    rrdemo
#9000 216500                LD    hl,101        ;Valeur initiale+1
                    loop00
#9003 2B                    DEC   hl            ;Compter
#9004 E5                    PUSH hl             ;Sauver l'etat du
                                                compteur
#9005 F7                    RST   #30           ;Prefixe de
                                                relocalisation
#9006 CD0D00                CALL numout-$       ;Sortir l'etat
                                                compteur
#9009 E1                    POP   hl            ;Restaurer l'etat
                                                compteur
#900A 7C                    LD    a,h
#900B B5                    OR    l             ;HL = 0 ?
#900C 20F5                  JR    nz,loop00     ;non: continuer a
                                                compter
#900E 3E0D                  LD    a,#0d         ;sortir CR
#9010 C35ABB                JP    txtout        ; TXT OUTPUT
                    ;
                    ; Sortir sur ecran le nombre 16 bits dans HL
                    ;
                    ; Registres BC,DE,HL,AF,IX, modifies
                    ;
                    numout
#9013 E5                    PUSH hl             ;Sauver le nombre a
                                                sortir
#9014 F7                    RST   #30
#9015 214B00                LD    hl,outbuf-$ ;Annuler le buffer
                                                de sortie
#9018 F7                    RST   #30
#9019 114800                LD    de,outbuf+1-$
#901C 010500                LD    bc,5
#901F 3620                  LD    (hl)," "
#9021 EDB0                  LDIR
#9023 E1                    POP   hl
#9024 F7                    RST   #30
#9025 DD213F00              LD    ix,outbuf+4-$
                    ;         IX indique le buffer de sortie
#9029 0E0A                  LD    c,10          ;Diviseur
                                                (constante)
#902B F7            loop01  RST   #30
#902C CD1D00                CALL div168-$       ;Diviser hl par 10
#902F C630                  ADD   a,"0"         ;Reste de la div
                                                ->chiffre ASCII
#9031 DD7700                LD    (ix+0),a      ;Ecrire chiffre
                                                dans le buffer
#9034 DD2B                  DEC   ix
#9036 EB                    EX    de,hl         ;Le quotien est le
                                                nouveau nombre
```

```
#9037 7C                  LD    a,h            ;Nombre
#9038 B5                  OR    l              ; = 0 ?
#9039 20F0                JR    nz,loop01      ;Non: continuer la
                                               convertion
#903B F7                  RST   #30
#903C 212400              LD    hl,outbuf-$    ;Sortir le buffer
#903F 0605                LD    b,5            ;Compteur de
                                               caracteres
#9041 7E        loop02    LD    a,(hl)         ;Prendre caractere
#9042 CD5ABB              CALL  txtout         ;Sortir caractere
#9045 23                  INC   hl             ;Prochain caractere
#9046 10F9                DJNZ  loop02         ;Tant que pas fini
#9048 C9                  RET                  ;Fin de la sortie
                ;
                ;
                ; Division 16 bits par 8 bits
                ; Entree: dividende dans HL, diviseur dans C
                ; Sortie: Quotient dans DE, reste dans A
                ; Registres modifies: AF,BC,DE,HL
                ;
#9049 110000    div168    LD    de,0           ;Preparer registre
                                               de quotient
#904C 0610                LD    b,16           ;Compteur de bits
#904E AF                  XOR   a              ; A=0
                loop03
#904F CB23                SLA   e              ;Quotient
#9051 CB12                RL    d              ;Decalage vers la
                                               gauche
#9053 CB25                SLA   l              ;Dividende
#9055 CB14                RL    h              ;Decalage vers la
                                               gauche
#9057 17                  RLA                  ;Bit de plus grande
                                               valeur dans A
#9058 B9                  CP    c              ;soustraction
                                               possible ?
#9059 3802                JR    c,skip00       ;Non: sauter
#905B 13                  INC   de             ;Augmenter quotient
#905C 91                  SUB   c              ;Soustraire diviseur
#905D 10F0      skip00    DJNZ  loop03         ;Tant que pas fini
#905F C9                  RET
                ;
                outbuf    DEFS  5              ;buffer de sortie
                                               de 5 octets
                txtout    EQU   #bb5a

Pass 2: 0 Errors

End of code:#9065
```

Dans le listing du programme de démonstration, vous trouvez de nouveau les adresses relatives préparées. Dans ce programme, toutes les instructions qui effectuent un adressage absolu dans le programme, sont précédées de RST &30 et leurs adresses sont transformées en adresses relatives au PC avec -$. Par exemple, les adresses qui concernent le tampon de sortie (OUTBUF aux adresses &0014, &0018, &0024 et &003C). Même l'utilisation d'un offset est sans problème avec le programme de relocation (p.ex. dans OUTBUF+4). Les instructions de chargement qui n'adressent pas le programme lui même (ex : le chargement de la constante à l'adresse &0000) ne sont pas modifiées. L'instruction LD IX,OUTBUF+1-$ à l'adresse &0024 démontre comment l'extension RST est utilisée avec les instructions qui ont un index-register-prefix. L'instruction RST est précédée par le préfixe. Le programme de relocation le reconnaît et en tient compte lorsqu'il traite l'instruction (voir les adresses &0005 à &0013 du programme de relocation).
Les instructions CALL du sous programme NUMOUT et DIV168 sont précédées d'un RST &30. Mais l'appel de la routine TXT OUTPUT n'est pas traité car il concerne le système d'exploitation et non le programme exemple.
De même, les instructions de branchement relatif JR et DJNZ ne sont pas traitées. Ce n'est pas nécessaire car elles ne concernent pas une adresse absolue.

La devise 'qui peut le plus peut le moins' n'est pas vraie pour le programme de relocation. Des instructions RST &30 inutiles modifient les codes qui leur succèdent et entraînent normalement le plantage du programme. Mais si on oublie un RST &30 dans le programme, il faut s'attendre au plantage lorsque le programme rencontrera l'instruction non traitée.

Nous en avons fini avec la théorie. Avec le programme BASIC qui suit vous pourrez charger le programme de démonstration :

```
10 input 'adresse de base';base
20 oldhimeme=himem
30 memory base-1
40 for mp=base to base+100
50 read octet
60 poke mp,octet
70 cs=cs+octet
80 next
90 if cs<>&2751 then print 'erreur dans les datas!':stop
100 call base
110 print'fini!'
120 memory oldhimem
130 end
1000 data &21,&65,&00,&2b,&e5,&f7,&cd,&0d
1005 data &00,&e1,&7c,&b5,&20,&f5,&3e,&0d
1010 data &c3,&5a,&bb,&e5,&f7,&21,&4b,&00
1015 data &f7,&11,&48,&00,&01,&05,&00,&36
1020 data &20,&ed,&b0,&e1,&f7,&dd,&21,&3f
1025 data &00,&0e,&0a,&f7,&cd,&1d,&00,&c6
1030 data &30,&dd,&77,&00,&dd,&2b,&eb,&7c
1035 data &b5,&20,&f0,&f7,&21,&24,&00,&06
1040 data &05,&7e,&cd,&5a,&bb,&23,&10,&f9
1045 data &c9,&11,&00,&00,&06,&10,&af,&cb
1050 data &23,&cb,&12,&cb,&25,&cb,&14,&17
1055 data &b9,&38,&02,&13,&91,&10,&f0,&c9
1060 data &00,&00,&00,&00,&00
```

Ce chargeur BASIC demande au début l'adresse de base voulue, réserve à cette adresse de la place mémoire et y copie le programme dedémonstration. Avant de refixer HIMEM avec l'instruction MEMORY, l'ancienne valeur de HIMEM est mémorisée et reprise à la fin du programme. Afin que ce programme puisse fonctionner il faut installer le programme de relocation. Sinon, cela conduirait certainement au plantage de l'ordinateur. Avec ce chargeur BASIC le programme peut être installé à n'importe quelle adresse libre. Il s'agit en particulier, compte tenu de la taille du programme BASIC, des adresses &1000 à &9fff. Il faut évidemment faire attention à ce que le programme de relocation ne soit pas écrasé par le programme de démonstration.

L'adresse maximum du programme de démonstration est donc l'adresse de base du programme de relocation (donnée par le chargeur de ce programme d'aide) moins la longueur du programme de démonstration (101 octets). Compte tenu de ces limites, le programme peut prendre n'importe quelle adresse de base. Si on supprime les instructions RST &30, il peut être utilisé comme un programme normal, à une adresse fixe. Vous constaterez en comparant les temps d'exécution que la rapidité n'est pratiquement pas modifiée.

Après cette analyse aproffondie du programme de démonstration, vous ne devriez éprouver aucune difficulté à manipuler le programme de relocation.

23.7.1 Les sous programmes du programme de démonstration

Comme nous l'avons indiqué dans la section précédente, nous allons parler plus particulièrement des sous programmes NUMOUT et DIV168. Evidemment, nous ne pouvons pas faire une analyse complète des routines arithmétiques, ce qui dépasserait le cadre de notre chapitre. Cependant, nous allons vous donner un aperçu des fonctions des routines afin que vous puissiez les utiliser vous même ou vous en inspirer pour vos propres routines. Nous espérons que cette description et les commentaires des routines dans le listing vous donnerons une bonne idée de leur fonction. Dans tous les cas, ces routines peuvent être utilisées telles qu'elles sont dans le listing (en particulier la routine de division) sauf indication contraire. Les commentaires dans le listing donnent des informations sur la façon de fixer les paramètres d'entrée et d'affichage. Les deux routines sont appelées par un simple CALL.

LA ROUTINE D'AFFICHAGE NUMOUT

Cette routine donne la valeur décimale du nombre binaire inscrit dans le registre HL sur 16 bits. L'affichage est fait sur cinq caractères à la position courante du curseur, Les chiffres non utilisés sont remplacés par des caractères vides. Il est donc possible d'afficher des valeurs de 0 à 65535.

La routine utilise pour l'affichage le tampon de sortie OUTBUF qui fait cinq caractères. Ce n'est pas seulement pour les préparer mais aussi parce que les nombres de cette routine sont construits à l'envers. Les chiffres sont d'abord écrits dans le tampon en commençant par le dernier puis affichés ainsi.

Au début de la routine d'affichage le tampon est rempli de caractères vides (adresses &0013 à &0023). Ceci est obtenu avec l'instruction LDIR. Avec la séquence d'instructions suivante :

```
LD HL,START
LD DE,START+1
LD BC,LENGTH-1
LD (HL),BYTE
LDIR
```

Un bloc de longueur LENGHT situé à l'adresse START peut être rempli avec la valeur BYTE car il se copie lui même. C'est une méthode relativement rapide (bien que ce ne soit pas la plus rapide) pour remplir un secteur mémoire avec une séquence d'octets constante.

Ensuite le registre IX est initialisé sur la fin du tampon comme pointeur et le registre C prend la valeur de la constante 10 utilisée pour la transformation.

Enfin, la transformation commence au label LOOP01. A cet effet, le nombre est divisé plusieurs fois par 10 (DIV168) et le reste obtenu est mémorisé dans le tampon (LD (IX),A) après une transformation en caractère ASCII (ADD A,'0'). Puis le pointeur de tampon est avancé (DEC IX). Le quotient naturel de la routine de division qui se trouve dans le registre DE est mémorisé comme nouveau nombre au début de la boucle dans HL (EX DE,HL). Ce procédé est repris jusqu'à ce que le quotient devienne nul après une boucle. Le test de nullité pour le registre 16 bit fait avec les instructions LD et OR est une méthode de test simple et rapide avec les registre HL, DE et BC. Elle est particulièrement valable pour les instructions 16 bits INC et DEC qui ne modifient pas les flags.

Si le quotient est nul, il n'y a plus de chiffre à traiter. La transformation est terminée et le nombre est inscrit dans le tampon. Le contenu du tampon est affiché à l'écran dans une boucle simple (LOOP02) à l'aide de la routine système TXT OUTPUT.

Pour mieux comprendre le travail de la routine d'affichage, voici l'exemple de la transformation du nombre 253 avec l'étude du contenu des registres et du tampon durant la boucle LOOP01 à l'adresse &0036. Les contenus des registres sont donnés en décimal, le contenu du tampon correspond aux caractères ASCII qui s'y trouvent. Le contenu du registre HL est donné non à l'adresse &0036 mais à l'adresse &002B, car il est modifié par la routine de division.

boucle	HL	DE	A	tampon
1	523	52	3	3
2	52	5	2	23
3	5	0	5	523

Après la troisième boucle, le quotient dans DE est nul et la boucle est terminée. A la fin de la routine l'instruction RET effectue le retour au programme principal.

LA ROUTINE DE DIVISION DIV168

Cette routine divise une nombre binaire sur 16 bits par un nombre sur 8 bits et donne le quotient sur 16 bits avec le reste sur 8 bits. Tous les nombres sont des nombres entiers non signés.

Afin de comprendre le fonctionnement de cette routine, il faut avoir une bonne connaissance du système binaire. Comme cela dépasse le cadre de ce chapitre, nous devrons nous en passer. Disons que cette routine de division fonctionne d'après le principe de la division écrite en binaire. Ne soyez pas déçus cependant de ne pas comprendre cette routine. La division binaire est une des routines les plus complexes en langage machine. Exécutez simplement la routine avec un exemple sur papier. Moi même, je n'ai compris la division binaire qu'après l'avoir programmée seul.

23.8. Les limites de la méthode de relocation

Dans cette section, nous allons donner les limites de la méthode de relocation que nous vous avons présentée.

Avec notre petit programme de relocation, nous pouvons avoir accès sans problème à un code objet dont l'adresse est fixée par l'opérande d'une instruction. Cela se complique lorsque nous voulons manipuler une structure de données, comme un tableau d'adresses par exemple. Mais ce problème lui aussi peut être résolu.

A cet effet, le programme en langage machine doit déterminer sa place en mémoire. Pour cela, il faut appeler ce netit sous programme :

```
GETPC  POP HL
       JP  (HL)
```

Après l'appel avec CALL GETPC, ce sous programme prend son adresse de retour sur la pile avec l'instruction POP et s'y branche avec l'instruction JP (HL). Cela à le même effet qu'une instruction RET mais charge l'adresse de retour, c'est à dire l'adresse du CALL se trouvant directement après ce sous programme, dans HL. Avec cela, nous avons tout ce qu'il nous faut. La séquence d'instructions :

```
        RST &30
        CALL GETPC-$
BASE    RST &30
        LD (BASADR-$),HL
```

permet de retrouver l'adresse absolue du label BASE. Nous pouvons donc entrer des données dans une table d'adresses relativement à BASE (pour des branchement p.ex.).

Au lieu de : DW adresse,

On écrit : DW adresse-BASE

dans la table.
Avant d'avoir accès à l'objet, il faut encore additionner l'adresse de BASE, qui se trouve dans la mémoire 16 bit BASADR, à l'adresse relative. Si l'adresse relative se trouve dans le registre HL p. ex., on peut exécuter la séquence suivante :

```
RST &30
LD DE,(BASADR-$)
ADD HL,DE
```

Les sous programmes et la mémoire utilisés pour cela prennent si peu de place qu'ils ne devraient pas poser de problèmes. Dans nos exemples, nous avons inscrit le préfixe (RST&30) où c'était nécessaire.

Avec les instructions que nous venons de donner, il est possible de rendre relogeables pratiquement tous les programmes qui utilisent des adresses branchant sur eux-mêmes. L'accès à un secteur d'adressage d'un autre programme est aléatoire car nous n'avons pas prévu la possibilité d'un accès global.

23.9 Le chargement d'un programme relogeable

Pour charger un petit programme, le chargeur BASIC est un moyen très pratique. Le chargeur BASIC d'un programme en langage machine peut être écrit sans problème par un autre programme. Le chargeur peut alors être ainsi constitué qu'il fixe l'adresse de base à partir de HIMEM comme c'est le cas pour le programme de relocation.

Pour des programmes longs, ce n'est pas très adéquat car la longueur du chargeur croît avec la taille du programme (un chargeur est environ trois fois plus long qu'un programme en langage machine). Pour cela, il faut un programme de chargement qui puisse lire un fichier binaire relogeable situé à l'adresse &0000. Dans ce cas aussi, l'établissement de l'adresse de base à partir de HIMEM est un système intéressant. Nous sommes en train de préparer un tel programme et il devrait être disponible avec la disquette de ce livre.

EAUTÉS NOUVEAUTÉS NOUV



LIVRE DU LECTEUR DE DISQUETTE AMSTRAD CPC (Tome 10)

Tout sur la programmation et la gestion des données avec le floppy DDI-1 et le 664 ! Utile au débutant comme au programmeur en langage machine. Contient le listing du DOS commenté, un utilitaire qui ajoute les fichiers RELATIFS à l'AMDOS avec de nouvelles commandes BASIC, un MONITEUR disque et beaucoup d'autres programmes et astuces... Ce livre est indispensable à tous ceux qui utilisent un floppy ou un 664 AMSTRAD.

Réf. : ML127
Prix : 149 FF



LE NOUVEL ATARI ST

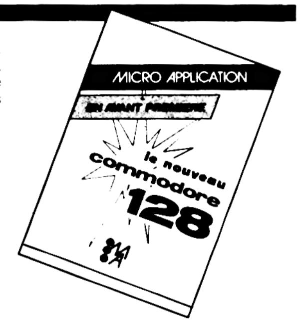
Ce livre décrit la superbe machine qu'est l'ATARI ST.
Architecture, interfaces, operating system, le bios, GEM, LOGO, le processeur 68000, sont quelques-uns des thèmes abordés. Ce livre doit être lu par tous ceux qui suivent de près le monde de la micro-informatique.

Réf. : ML125
Prix : 129 FF

LE NOUVEAU COMMODORE 128

Ce livre présente le nouveau Commodore 128. Vous y trouverez un aperçu complet des possibilités du successeur du célèbre "64" et une présentation détaillée des trois operating system. Le super nouveau BASIC Commodore 7.0 est décrit ainsi que la configuration de la mémoire, la page zéro et le nouveau et rapide lecteur de disquette 1571. Pour tous les Commodoristes !

Réf. : ML130
Prix : 129 FF



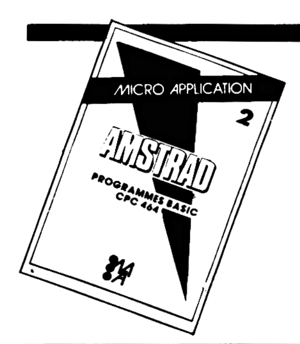
LES LIVRES AMSTRAD



TRUCS ET ASTUCES POUR L'AMSTRAD CPC (Tome 1)

C'est le livre que tout utilisateur d'un CPC doit possèder. De nombreux domaines sont couverts (graphismes, fenêtres, langage machine) et des super programmes sont inclus dans ce best-seller (gestion de fichiers, éditeur de texte et de sons...).

Réf. : ML112
Prix : 149 FF



PROGRAMMES BASIC POUR LE CPC 464

ALIMENTEZ VOTRE CPC 464

Ce livre contient de super programmes, notamment un désassembleur, un éditeur graphique, un éditeur de texte... Tous les programmes sont prêts à être tapés et abondamment commentés.

Réf. : ML119
Prix : 129 FF

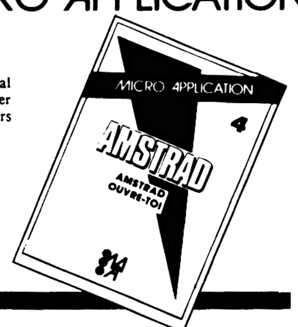
LE BASIC AU BOUT DES DOIGTS CPC 464

Ce livre est une introduction complète et didactique au BASIC du micro-ordinateur AMSTRAD CPC 464. Il permet d'apprendre rapidement et facilement la programmation (ins-tructions BASIC, analyses des problèmes, algorythmes complexes...)
Principaux thèmes abordés :
- Les bases de la programmation
- Bit, Octet, ASCII
- Instructions du BASIC
- Organigrammes
- Les fenêtres
- Programmes BASIC plus poussés
- Le programme et menus.

Comprenant de nombreux exemples, ce livre vous assure un apprentissage simple et efficace du BASIC CPC 464.

Réf. : ML118
Prix : 149 FF

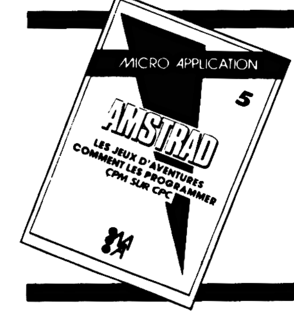




AMSTRAD OUVRE-TOI

Le bon départ avec le CPC 464 ! Ce livre vous apporte les principales informations sur l'utilisation, les possibilités de connexions du CPC 464 et les rudiments nécessaires pour développer vos propres programmes. C'est le livre idéal pour tous ceux qui veulent pénétrer dans l'univers des micro-ordinateurs avec le CPC 464.

Réf. : ML120
Prix : 99 FF



JEUX D'AVENTURES. COMMENT LES PROGRAMMER

Voici la clé du monde de l'aventure. Ce livre fournit un système d'aventures complet, avec éditeur, interpréteur, routines utilitaires et fichiers de jeux. Ainsi qu'un générateur d'aventures pour programmer vous-mêmes facilement vos jeux d'aventures. Avec, bien sûr, des programmes tout prêts à être tapés.

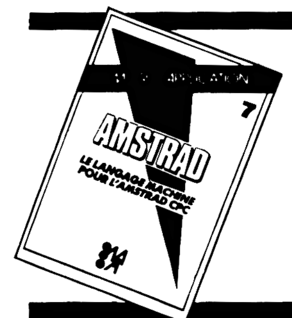
Réf. : ML121
Prix : 129 FF



LA BIBLE DU PROGRAMMEUR DE L'AMSTRAD CPC 464
(Tome 6)

Tout, absolument tout sur le CPC 464. Ce livre est l'ouvrage de référence pour tous ceux qui veulent programmer en pro leur CPC. Organisation de la mémoire, le contrôleur vidéo, les interfaces, l'interpréteur et toute la ROM DESASSEMBLEE et COMMENTEE sont quelques-uns des thèmes de cet ouvrage de 700 pages.

Réf. : ML122
Prix : 249 FF



LE LANGAGE MACHINE DE L'AMSTRAD CPC (Tome 7)

Ce livre est destiné à tous ceux qui désirent aller plus loin que le BASIC. Des bases de la programmation en assembleur à l'utilisation des routines système, tout est expliqué avec de nombreux exemples. Contient un programme assembleur, moniteur et désassembleur.

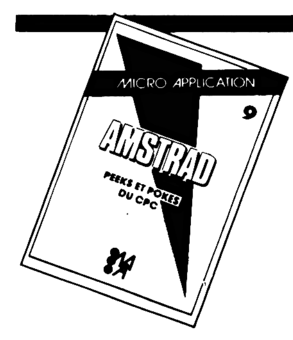
Réf. : ML123
Prix : 129 FF

GRAPHISMES ET SONS DU CPC

L'AMSTRAD CPC dispose de capacités graphiques et sonores exceptionnelles. Ce livre en montre l'utilisation à l'aide de nombreux programmes utilitaires.
Contenu :
- base de programmation graphique
- éditeur de police de caractères
- "sprites", "shapes", et chaines
- représentations multi-couleurs.
- calcul des coordonnées
- rotations, mouvements
- représentations graphiques de fonctions en 3D
- D.A.O. (dessin assisté par ordinateur)
- synthétiseur
- mini-orgue
- enveloppes de son, et beaucoup d'autres choses...

Réf. : ML124
Prix : 129 FF





PEEKS ET POKES DU CPC
(Tome 9)

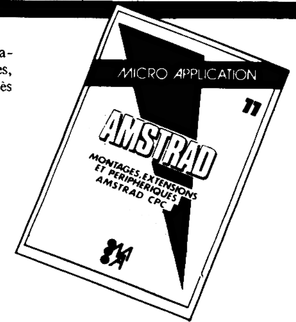
Comment exploiter à fond son CPC à partir du BASIC ? C'est ce que vous révèle ce livre avec tout ce qu'il faut savoir sur les peeks, pokes et autres call... Vous saurez aussi comment protéger la mémoire, calculer en binaire... et tout cela très facilement. Un passage, assuré et sans douleur du BASIC au puissant LANGAGE MACHINE.

Réf. : ML126
Prix : 99 FF

MONTAGES, EXTENSIONS ET PERIPHERIQUES AMSTRAD CPC
(Tome 11)

Pour tous les amateurs d'éléctronique ce livre montre ce que l'on peut réaliser avec un CPC. De nombreux schémas et exemples illustrent les thèmes et applications abordés comme les interfaces, programmateur d'EPROM... Un très beau livre de 450 pages.

Réf. : ML131
Prix : 199 FF



LE LIVRE DU CP/M AMSTRAD (Tome 12)

Ce livre vous permettra d'utiliser CP/M sur les CPC 464, 664 et 6128 sans aucune difficulté. Vous y trouverez de nombreuses explications et les différents exemples vous assureront une maitrise parfaite de ce très puissant système d'exploitation qu'est CP/M. (300 pages).

Réf. : ML128
Prix : 149 FF





DES IDEES POUR LES CPC (Tome 13)

Vous n'avez pas d'idées pour utiliser votre CPC (464, 664, 6128) ? Ce livre va vous en donner ! Vous trouverez de très nombreux programmes BASIC couvrant des sujets très variés qui transformeront votre CPC en un bon petit génie. De plus les programmes vous permettront d'appronfondir vos connaissances en programmation.
(250 pages).

Réf. : ML132
Prix : 129 FF

AMSTRAD AUTOFORMATION A L'ASSEMBLEUR EN FRANCAIS

Contient un livre et un logiciel.
LE LIVRE :
Cet ouvrage introduit le débutant à la programmation du Z80 grâce à la méthode du DR WATSON qui selon les critiques vaut son pesant d'or ! Aucune connaissance préalable n'est requise et le but du livre est d'assurer au novice un succès total. A la fin du livre les instructions du Z80 sont expliquées en détail. De nombreux exemples illustrent les différentes étapes du cours alors que des exercices (les solutions sont fournies) testent la compréhension.
LE LOGICIEL : Un assembleur Z80 complet est livré sur cassette et comprend :
- Etiquettes Symboliques
- Directives d'Assemblage
- Chargement/Sauvegarde
- Copie Ecran
- INSERT / DELET.

L'assembleur permet d'écrire des programmes facilement en langage d'assemblage puis les transforme en code machine (langage machine). Pour vous aider à comprendre les rotations mathématiques utilisées, une démonstration de l'utilisation des nombres binaires et hexadécimaux est fournie. Un programme utilisant les commandes graphiques additionnelles décrites dans le livre est également fourni.

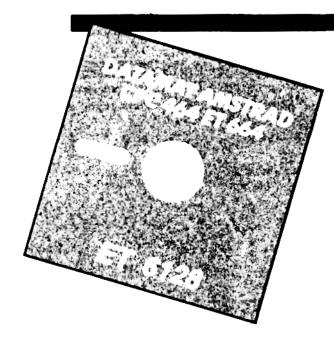
Réf. : ML126
Prix : 195 FF K 7 - 295 FF - disquette

TEXTOMAT AMSTRAD
CPC 464 & 664

Traitement de texte de qualité professionnelle pour tous.
Tabulation, recherche, remplacement, insertion, manipulation de paragraphes, calcul... Accents à l'écran et imprimante. Module permettant de gérer tout type d'imprimante. Ecrit en LANGAGE MACHINE. Liaison avec DATAMAT pour mailing et lettres types personnalisées... TEXTOMAT est la solution traitement de texte sur CPC. Documentation complète.

Réf. : AM305
Prix : 450 FF





DATAMAT AMSTRAD
CPC 464 & 664

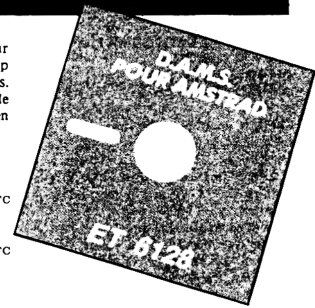
La gestion de fichier la plus complète fonctionnant pour les 464 et 664. Entièrement en LANGAGE MACHINE. Fonctions de calcul, de tri, de recherche multicritères, impressions paramètrables, liaison avec TEXTOMAT pour mailing... Documentation française de 60 pages.

Réf. : AM304
Prix : 450 FF

D.A.M.S. POUR AMSTRAD
CPC 464 & 664

D.A.M.S. est un logiciel intégrant un assembleur, un moniteur et un désassembleur symbolique pour développer et mettre au point facilement des programmes en langage machine sur les micro ordinateurs AMSTRAD. Les trois modules sont co-résidents en mémoire ce qui assure une grande souplesse d'utilisation. Vous pouvez notamment utiliser un éditeur plein écran, un assembleur immédiat, un désassembleur symbolique, une trace et beaucoup d'autres fonctions très puissantes. D.A.M.S. est entièrement relogeable et est bien évidemment écrit en langage machine.

Réf. : AM208
Prix : sur cassette : 295 FF TTC pour CPC 464
Réf. : AM308
Prix : sur disquette : 395 FF TTC pour CPC 664 & CPC 464




MICRO INFO
123
XB2
ATARI
80 pages de trucs et astuces, programmes, dossiers pour Amstrad CPC, Commodore Atari ST ...
20 f

Achevé d'imprimer en janvier 1986
sur les presses de l'imprimerie Laballery et Cie
58500 Clamecy
Dépôt légal : janvier 1986
Numéro d'imprimeur : 601094

IMPRIMÉ EN FRANCE

[redacted]

Ce livre concerne tous les possesseurs de CPC (464, 664, et bien sûr 6128). Vous y trouverez un générateur de menus, un générateur de masques, des aides à la programmation comme un DUMP, l'utilisation des routines systèmes et plein d'astuces de programmation. Pour tous ceux qui veulent tirer le maximum de leur CPC !

Contenu :

- programmes de tri
- graphisme tridimensionnel
- générateur de menus
- générateur de masques de saisie
- protection de vos propres programmes
- DUMP des variables
- impression de l'écran graphique
- création de lignes de BASIC à partir du BASIC
- des conseils indispensables pour programmer en langage machine
- scrolling en douceur
- utilisation des registres du Z80 en BASIC
- routines de l'interpréteur et du système d'exploitation
- compatibilité entre les 3 CPCs
- programmes en langage machine relogeables

Prix : 129F TTC
Réf. : ML147
ISBN : 2-86899-038-X


AMSTRAD
TRUCS ET ASTUCES II
POUR LES CPC 464, 664 ET 6128
17
MA

TRUCS ET ASTUCES II POUR CPC (tome 17)

Ce livre concerne tous les possesseurs de CPC (464, 664 et bien sûr 6128 !). Vous y trouverez un générateur de menus, un générateur de masques, des aides à la programmation comme un DUMP, l'utilisation des routines systèmes et plein d'astuces de programmation. Pour tous ceux qui veulent tirer le maximum de leur CPC !

Réf. : ML 147

Prix : **129** F TTC

